MASTER EN SCIENCES INFORMATIQUES

Développement systématique d'une librairie de spécifications algébriques et de son implémentation en Prolog

Titeca, Eric

Award date:
1990

Awarding institution:
Universite de Namur

Facultés Universitaires
Notre-Dame de la Paix
Institut d'informatique
NAMUR

Développement systématique d'une librairie de spécifications algébriques et de son implémentation en Prolog.

Eric TITECA

Mémoire présenté en vue de
l'obtention du titre de Licencié
et Maître en Informatique

Promoteur: A. VAN LAMSWEERDE
Assistance : N. HABRA

Année académique 1989-1990

RESUME

Facultés Universitaires Notre-Dame de la Paix
Institut d'informatique
rue de Bruxelles 61, B-5000 NAMUR
Tél. 081-22.90.61 Télex 59222 facnam-b Téléfax 081-23.03.91

Développement systématique d'une librairie de spécifications algébriques et de son implémentation en Prolog.

TITECA Eric

Résumé.

L'emploi de procédés méthodiques de spécification et de codage de procédures permet d'augmenter la fiabilité et la viabilité d'un produit logiciel. Dans cette optique, nous avons développé une méthodologie en trois étapes, utile à la conception de procédures Prolog exécutables.

La première adapte le mécanisme d'induction structurelle afin d'établir, de façon cohérente et complète, les spécifications en termes de types abstraits algébriques. La transformation de ces spécifications sous forme de relations définies en logique des prédicats du premier ordre est décrite par la deuxième étape. Elle permet d'obtenir une seule procédure logique d'évaluation de fonctions, extensible à volonté. Enfin, en tenant compte du mécanisme d'exécution du langage Prolog et du mode d'utilisation souhaité des traitements, l'évaluateur est codé et peut éventuellement être décomposé en procédures Prolog relationnelles (afin d'obtenir deux interfaces différentes).

Chaque étape est critiquée, et une intégration de ce mémoire dans un projet plus vaste de prototypage est explicité.

Mémoire de licence et maîtrise en Informatique
Juin 1990
Promoteur : Prof. A. VAN LAMSWEERDE
Assistance : N. HABRA

Facultés Universitaires Notre-Dame de la Paix
Institut d'informatique
rue de Bruxelles 61, B-5000 NAMUR
Tél. 081-22.90.61 Télex 59222 facnam-b Téléfax 081-23.03.91

Systematic development of an algebraïc requirements library and its implementation in Prolog.

TITECA Eric

Abstract.

The use of systematic methods of specification and implementation to obtain runnable procedures can increase reliability and life of a software product. With this point of view, we have developped a methodology in three steps, useful for the conception of Prolog procedures.

The first one fit the structural induction mechanism in order to construct coherent and complete specifications in term of algebraïc abstract data types. Transformation of this specifications as a set of relations defined in the first order predicate logic is described by the second step. It gives only one extensible logic procedure that evaluates all the fonctions. At last, Prolog language running mechanism and treatment mode of use in one hand, the evaluator is implemented, and can eventually be splitted into relational Prolog procedures (in order to obtain two different interfaces).

Criticism of every step is presented, and work integration in a larger prototype project is explicited.

Mémoire de licence et maîtrise en Informatique
Juin 1990
Promoteur : Prof. A. VAN LAMSWEERDE
Assistance : N. HABRA

REMERCIEMENTS

Au terme de ce travail, qu'il me soit permis de remercier certaines personnes sans lesquelles je n'aurais sans doute jamais connu la spécification abstraite de la même manière.

Merci à Mr. le professeur Axel VAN LAMSWEERDE, pour avoir accepté d'être mon promoteur et pour son accueil dans l'unité de recherche qu'il dirige.

Mes remerciements vont tout particulièrement à Mr. Nagy HABRA. Grâce à sa grande disponibilité, sa franche sympathie et ses nombreux conseils judicieux, il fut pour moi le guide indispensable à la réussite du mémoire.

Mr. Yves DEVILLE, par ses conseils éclairés, a grandement contribué à ce mémoire.

Je m'en voudrais d'oublier la personne qui m'a forcé à acquérir une rigueur qui m'échappe bien souvent, qui m'a ouvert les yeux dans bien des domaines, même extra-académiques : Mr. Philippe VAN BASTELAER.

Mille mercis aux membres de l'institut, académiques et scientifiques, avec lesquels j'ai passé une année agréable et enrichissante.

Je désirais tant nommer une dernière personne, mais celle-ci me l'a refusé, par pudeur d'esprit. Quoiqu'il en soit, elle se reconnaîtra.

SOMMAIRE

INTRODUCTION

Il est rare qu'un logiciel informatique de taille appréciable ne présente jamais d'erreurs de fonctionnement; erreurs que les informaticiens chargés de la maintenance peuvent localiser à toutes les étapes de la conception du logiciel.

C'est pourquoi les analystes-programmeurs s'intéressent de plus en plus aux outils d'aide à la programmation, tels les éditeurs syntaxiques et les compilateurs évolués de langages de haut-niveau ayant des analyseurs sémantiques. Mais un des aspects fondamentaux de ces outils reste le développement de méthodes adaptées aux besoins des concepteurs.

La grande difficulté de la conception de logiciel reste l'étape primordiale sur laquelle repose la fiabilité de l'édifice logiciel. Pratiquement tous les ingénieurs logiciels sont d'accord pour dire qu'il s'agit de la phase de spécification du problème à résoudre par l'informatique.

Pour faire face à cette difficulté, il serait aisé et utile que l'analyste-programmeur dispose de librairies de spécifications vérifiées, avec, éventuellement, les procédures correspondantes dans un langage cible donné. L'agencement original de ces spécifications (puis de ces procédures), selon les besoins d'une nouvelle architecture, ainsi que l'adjonction de spécifications neuves et spécifiques et de leurs procédures correspondantes, pourraient rapidement former le logiciel attendu. Ces dernières sont, à leur tour, susceptibles d'être insérées dans les librairies existantes.

Ce type de méthodologie de construction de logiciels rassemble en un mouvement alterné, les méthodes dites "Top-Down" ou descendante, et "Bottom-Up" ou ascendante.

La première méthodologie développe une architecture modulaire par raffinements successifs de la fonction d'abstraction du système. Quant à la seconde, elle détermine, par effet de regroupement et de généralisation, un ensemble de besoins de base aux propriétés communes.

Nous avons fondé ce travail dans cette dernière optique : la conception de telles librairies de spécifications réutilisables et suffisamment générales que pour s'adapter à différents contextes. De plus, le caractère extensible de ces ensembles de spécifications et/ou de procédures doit apparaître clairement. L'idée motrice est de bâtir une librairie autour de laquelle des sytèmes spécifiques graviterons, et puiserons la spécification de leurs besoins généraux.

La première phase du travail fut la détermination des opérations à spécifier. Notre choix repose sur les objets génériques à manipuler par la plupart des systèmes logiciels.

Deux grandes classes d'objets sont apparues : d'une part, ce sont les objets de type simple, intrinsèque, découlant des mathématiques et de l'écriture, tels les booléens, les naturels, les entiers et réels, et les caractères. D'autre part, il y a certaines structures indispensables par leur rôle d'ordonnancement particulier d'autres types, appelés types paramétrés, tels la liste, l'union ou le produit cartésien. Les instances ordonnancées par ces structures génériques sont les objets de ces types.

Suite à cette première phase est obtenue une liste des opérations principales et nécessaires, représentative de chaque type d'objet.

Dans une deuxième phase, il nous fallut retenir une méthode de conception et de description, à la fois syntaxique et sémantique, de chaque opération portant sur les objets choisis. La technique des types de données abstraits, méthode formelle, fiable et vérifiable, a paru être suffisamment simple, de par sa relative simplicité d'écriture et d'expression (utilisant le mécanisme de récursivité). Elle fait l'objet du premier chapitre.

La littérature offrant nombre de formalismes de cette technique, nous en avons redéfini un, reprenant les caractéristiques de la plupart, et déterminant le langage de description des axiomes équationnels de types de données abstraits.

Ce formalisme, que nous décrivons au second chapitre, est fondé sur le langage des prédicats du premier ordre. Il s'adapte à merveille au mécanisme de récursivité que prône la technique de spécification sémantique des types abstraits.

La troisième phase a permis de concevoir, de façon correcte, les axiomes spécifiant chaque opération des librairies. Nous nous sommes inspirés de travaux dans le domaine de l'obtention d'algorithmes logiques, corrects par construction (DEVILLE '87) afin de définir une démarche débouchant, sans démonstration ultérieure, sur des axiomes équationnels exacts. Cette démarche permet à chaque analyste d'écrire rapidement, sans erreur, des règles définissant une opération de manière univoque. Elle est explicitée dans le troisième chapitre.

Le quatrième chapitre, quant à lui, trace brièvement les grandes lignes du langage de programmation choisi pour l'implémentation de chaque opération en un ensemble de procédures exécutables, fiables et adaptables, qui forment le programme qu'attend le demandeur.

En effet, dans un but double d'exécution et de vérification des spécifications, afin de tester leur satisfaisabilité, nous avons développé une quatrième phase de notre travail. Le langage choisi pour celle-ci devait être intéressant de par sa rapidité de mise en oeuvre, et de par sa proximité avec le langage de description des axiomes. De plus, les besoins de prototypage ne demande, en aucune manière, que le langage soit efficace. Le Prolog, langage de programmation logique, a été le langage judicieux retenu.

Puisque la transformation des axiomes spécifiés en prédicats du premier ordre est systématique pour chacun d'entre-eux, cette phase a permis de mettre en évidence des règles successives de transformation quasi-automatique des axiomes en procédures Prolog éxécutables. Le cinquième chapitre du travail décrit cet ensemble de règles, avec la démonstration de la validité théorique de chacune, dans un souci de cohérence.

Ce travail ne serait intéressant s'il ne comportait pas l'explication des problèmes rencontrés lors de l'application de la méthodologie, et des solutions apportées. Les deux étapes qui la composent sont analysées dans les deux derniers chapitres.

Enfin, un court chapitre, précédant les conclusions de ce mémoire, présente quelques aspects futurs pouvant être explorés, principalement dans le cadre d'une automatisation des deux démarches décrites, et dans l'optique d'un parallélisme avec la méthode de spécification orientée-objet.

L'annexe de ce mémoire comprend la liste des spécifications en type de données abstraits et des procédures Prolog découlant de leur transformation systématique, pour quelques opérations de chacun des types de données retenus.

PREMIERE PARTIE : LA SPECIFICATION ABSTRAITE DE PROBLEMES

Face à un problème d'une certaine taille, que l'on voudrait résoudre par l'informatique, l'analyste doit disposer de méthodes de structuration et de spécification des besoins qui lui permettront de concevoir une documentation apte à être programmée dans un langage quelconque.

Deux méthodes, complémentaires, de construction d'une architecture logicielle sont envisageables.

Il s'agit, d'abord, de la méthode descendante ("Top-Down"), qui développe une architecture par raffinements successifs de la fonction d'abstraction du système. Cela signifie, autrement dit, que l'analyste décompose le problème en sous-problèmes plus simples. Ceux-ci sont liés par une relation de hiérarchie avec le problème de base, et par une relation de succession entre eux. Cette méthode est récursive, chaque sous-problème subissant à son tour la décomposition, et ceci jusqu'à l'obtention de sous-problèmes connus, soit déjà résolus (spécifiés dans une librairie, par exemple), soit suffisamment simples que pour être spécifiés par une autre méthode. Ce type de méthodologie a été envisagé dans les travaux de notre unité de recherche (DUBOIS '87, DEMEULEMEESTER-LALOY '89, ...) et permet d'obtenir rapidement des prototypes satisfaisant tout ou une partie de la demande.

La seconde méthode, dite ascendante ("Bottom-Up"), considère, quant à elle, la solution du plus grand nombre, et de la structuration en fonction des besoins. Utilisant des méthodes de spécification et de programmation de problèmes de petite taille, elle établit une librairie de solutions qui seront agencées selon les besoins ultérieurs, inconnus à ce moment. Le désavantage qui serait dû à l'effort inutile de certaines spécifications est contrebalancé par le choix subtil de la généralité des problèmes à résoudre.

Soulignons la complémentarité des deux méthodes : la méthode "Top-Down" décante le problème du demandeur jusqu'au niveau de la librairie définie dans ce travail. Dans le cadre du prototypage - dont la définition est donnée par Habra '90 - étudié dans l'unité de recherche, c'est cette dernière méthode qui a été explorée dans ce travail. Après l'isolement de besoins revenant régulièrement dans les travaux précédents (Demeulemeester-Laloy ('89), Dubois ('76), ...), il a fallu déterminer une méthode globale de spécification de ces besoins. C'est l'objet de cette première partie qui expose les techniques et les outils de spécification retenus.

CHAPITRE 1 : Ecriture de spécifications par la méthode des types de données abstraits

La méthode de description des types de données abstraits (**T.D.A.**) est une méthode de spécification d'ensembles d'objets dont la syntaxe et la sémantique sont définies formellement grâce à des bases théoriques mathématiques rigoureuses.

Ainsi, construire la représentation abstraite d'un ensemble d'objets de même type, dans un système formel choisi, revient à énoncer un certain nombre de formules valides de ce système, caractérisant ce type. La validité de chacune d'elles est obtenue par démonstration de l'exactitude, que DEVILLE ('87) et HABRA ('89) décrivent mathématiquement.

L'objectif de ce chapitre est de. rendre les chapitres suivants accessibles à ceux qui ne seraient pas habitués aux spécifications algébriques. Une description mathématique rigoureuse des types de données abstraits est donnée par HABRA ('90), et, il n'est pas dans notre volonté d'aller jusqu'à ce niveau. Seules les explications nécessaires et suffisantes à ce travail seront reprises.

A. Avantages et inconvénients de la spécification en TDA.

Nous avons choisi d'exprimer les spécifications formelles de la librairie d'opérations primitives à l'aide des TDAs, car le processus de conception peut être entrepris sans aucune référence à une représentation explicite des objets, dont l'implémentation de ceux-ci peut varier selon les ordinateurs. De plus, mais cet aspect ne nous concerne que peu, les TDAs sont un outil puissant de programmation (SIMULA ou CLU), de par leur facilité d'utilisation et leur fiabilité logicielle.

L'expression de types de données est équationnelle. Les équations seront exploitées par le processus d'évaluation des TDAs : la réécriture. Il s'agit du mécanisme de simplification d'un terme par d'autres termes, plus simples, formant une égalité de termes (pour le lecteur pressé, La notion formelle de terme intervient plus loin).

Les TDAs offrent de nombreux avantages dans le processus de spécifications par décompositions successives et réécritures, dont les principaux sont :

- L'expression en TDA est suffisante,
- Lors du processus de réécriture, on fait apparaître explicitement les relations implicites entre un objet et ses composants,
- Certaines propriétés non fournies par le spécifieur apparaissent et permettent quelques validations utiles, et,
- La réécriture n'est qu'une reformulation préservant toute la sémantique de la spécification.

Les types de données abstraits ont d'autres avantages, plus techniques, que DUBOIS ('87) relève : ce sont

- l'application aisée de traitements automatiques très sophistiqués (cohérence, complétude ou maquettage, par exemple), et,
- la vérification facile de la correspondance entre le programme et sa spécification.

L'expression de spécification en TDA offre peu d'inconvénients. Ceux-ci sont surmontables par un court apprentissage de cette technique. En voici trois :

- Le problème de communicabilité entre non habitués, qui existe dans l'apprentissage de tout nouveau langage,
- le pouvoir limité d'expression dû à une formulation peu naturelle, et,
- la faible concision de l'expression de certains problèmes, par rapport aux spécifications non formelles.

Ces deux derniers inconvénients sont compensés par une documentation claire et parsemée de commentaires informels, mais immédiatement compréhensibles.

B. Le mécanisme d'abstraction des données (ou objets).

Le mécanisme d'abstraction des données est basé sur les types de données abstraits. Un objet est exprimé, soit à l'aide d'axiomes caractérisant des opérations définies dans un ensemble de données (ce sont les TDAs), soit en une famille de TDAs structurée par des procédés d'abstraction.

Nous nous attacherons au premier cas, l'autre étant amplement développé dans le mémoire de DEMEULEMEESTER et LALOY ('89).

1. Les spécifications axiomatiques de données formelles

LISKOV et ZILLES ('74) définissent un Type de Donnée Abstrait (TDA) comme, je cite, "un ensemble fini d'objets capables de comportements particuliers, limités à un ensemble fini d'opérations permises sur des objets de ce type". De là, toutes les autres opérations seront réalisées utilisant ces opérations permises.

a. Définition d'un Type de Donnée Abstrait.

Le TDA est défini par trois éléments, outre le nom qui l'identifie :

- **Set**, l'ensemble des sortes internes, ou encore, l'ensemble fini des éléments que les opérations manipulent (domaine) ou créent (codomaine). Cet ensemble contient le sous-ensemble des éléments du type à spécifier (Sorte d'intérêt). Set est encore appelé l'environnement de la spécification.

- **SIGMA**, la signature du TDA, est l'ensemble des opérations permises sur la sorte d'intérêt.

* Chaque opération est une fonction caractérisée par

- son nom, OMEGA étant l'ensemble des noms d'opérations de tel TDA,
- son profil, qui reprend le domaine, ou sortes reçues en arguments, et le codomaine, ou "la" sorte fournie en résultat (Un seul résultat par opération !).

* Les opérations sont classées en deux catégories.

D'abord les Constructeurs, ou générateurs, qui produisent des objets de la sorte d'intérêt en résultat. Ces opérations modifient, construisent ou génèrent cette sorte.

Les contructeurs non primordiaux, c'est-à-dire ceux qui ne sont pas strictement nécessaires à la construction de la sorte d'intérêt, sont appelés, préférentiellement, les modificateurs.

Ensuite, nous avons les Sélecteurs, ou fonctions d'accès, qui déterminent une propriété de la sorte. Celle-ci est un argument de ce type de fonction, mais jamais un résultat.

- **AX**, pour axiomes, qui comprend toutes les équations auxquelles doivent satisfaire les opérations permises. Les équations sont des relations d'équivalence entre deux termes. Chaque équation permet de simplifier ou de représenter la fonction abstraite en termes de quelques constructeurs de base, ou d'autres fonctions qui sont elles-mêmes représentés de cette façon (application de la méthode "Top-Down").

Le sous-ensemble des constructeurs générateurs de la sorte d'intérêt n'a pas à être spécifié, ces derniers étant implicites. Seule une explication textuelle de leur but est fournie, permettant ainsi certaines options d'implémentation (de plus amples détails seront fournis au chapitre 3).

Il est clair qu'une opération ne peut pas se déterminer à l'aide d'un ensemble d'autres, si au moins une de ces autres opérations se détermine par cette opération. Les définitions cycliques sont indémontrables, et DEVILLE ('87) explique que si cela était permis, l'exactitude ne serait plus monotonique et empêcherait toute démonstration.

Remarque : Le plus délicat dans la description d'un type semble être la détermination de l'ensemble minimal et suffisant des constructeurs de base.

b. Remarque sur la notion d'algèbre

Puisqu'une famille d'opérations est liée à l'ensemble des données à spécifier, les TDAs sont des algèbres. Cette notion est détaillée formellement par HABRA ('89).

Les TDAs sont composés d'une partie syntaxique (signature), et d'une partie sémantique représentée par les axiomes.

c. Le formalisme de description des TDAs

La littérature offre nombre de formalismes différents pour exprimer les axiomes d'un TDA (équation, conditionnelle ou non, clauses de Horn avec égalité ou non, logique des prédicats du premier ordre, ...).

HABRA ('89) précise que la satisfaction offerte par chaque approche diffère pour le même ensemble d'opérations, et que, de plus, les règles habituelles d'inférence ou de construction des équivalences sont différentes.

Notre démarche s'inspire de la thèse de DEVILLE ('87), concernant la construction d'algorithmes logiques corrects. Elle se base sur la logique des prédicats du premier ordre, que nous présentons au chapitre suivant. Nous présentons toutefois quelques exemples tirés de la littérature, basés sur d'autres formalismes.

2. Spécification de la syntaxe d'un TDA.

L'ensemble des sortes internes, S, énumère ces sortes par leur nom. Habituellement, la sorte d'intérêt a le même nom que le type, et est la première de la liste, afin de la mettre en évidence. Les autres sortes sont considérées comme préspécifiées correctement.

On ordonne la signature du type de cette façon : viennent d'abord les constructeurs primordiaux. Suivent ensuite les autres constructeurs, et enfin les sélecteurs.

a. Distinction type paramétré ou non

Les types de données abstraits peuvent être classés en types paramétrés et types non paramétrés.

Déf. Un *type paramétré* est un type possédant au moins un paramètre formel représentant n'importe quel type. Ces paramètres formels seront remplacés par un type déterminé lors de l'utilisation du type paramétré.

Les types non paramétrés sont des ensembles d'éléments aux caractéristiques abstraites communes, dont tous les paramètres sont typés. Le mécanisme permettant de passer d'un type paramétré à un type non paramétré est l'instanciation.

Exemple : Seq(X) ----------> Seq(Int)
instanciation

b. Construction de la syntaxe d'un TDA

La première partie d'une spécification par les TDAs est la détermination de la syntaxe correcte du type à définir, et dont l'aboutissement est l'obtention de spécifications ayant deux objectifs : servir de base à la construction de la sémantique d'un TDA, et d'autre part, servir d'interfaces utilisateurs.

Nous promulguons, ci-dessous, notre propre syntaxe abstraite, qui reprend quelques caractéristiques communes à plusieurs formalismes. L'élaboration de cette syntaxe a poursuivi le triple but de :

- clarté,
- simplicité, concision, et,
- richesse des spécifications,

afin de satisfaire au critère de rapidité et d'exactitude de communication des spécifications.

La forme générale d'une spécification syntaxique d'un type de données abstrait est présenté par le tableau ci-dessous :

MODULE NOM_DE_TYPE

TYPE DE DONNEES : nom_de_type

Inf : Un objet de type nom_de_type est . . .

Set : set_courant, sets_utiles.

Sigma :

Cons	:	op1	:	dom1	-->	codom1
		...		...		...
		opI	:	domI	-->	codomI
Modif	:	opJ	:	domJ	-->	codomJ
		...		...		...
		opK	:	domK	-->	codomK
Sélec	:	opL	:	domL	-->	codomL
		...		...		...
		opN	:	domN	-->	codomN

Le point 1 nomme le type de la structure de donnée que l'on veut spécifier (nom_de_type).

Une description informelle (Inf) précise brièvement au point 2, et le plus complètement possible, ce qu'est un objet de tel type. C'est en réalité la propriété générale, ou la relation, exprimée textuellement, de cette classe d'objets. Ceci est une facilité permettant de comprendre, sans longue réflexion, le type, à l'aide d'informations supposées connues.

Au point 3, Il y a énumération des différents ensembles de données (Set, ou sortes) que recevront en arguments ou fourniront comme résultat les opérations possibles sur cette structure. Le premier ensemble est l'ensemble de la sorte interne, c'est-à-dire celui qui forme le type à spécifier.

Enfin, pour chaque opération, on décrit la SIGNATURE sigma (point 4).

- détermination de la classe de l'opération :

Constructeur	:	fournit un élément du type à décrire,
Modificateur	:	change l'état d'un élément du type,
Sélecteur	:	fournit un élément de type différent de celui à décrire.

- dénomination de l'opération
- énumération des arguments par leur set (DOMAINE, dom_i)
- énumération du résultat par son set (CODOMAINE, $codom_i$)

Un léger désavantage est la restriction de la spécification à des opérations n'ayant qu'un unique résultat. Cela est résolu aisément par l'emploi d'une succession d'opérations pour obtenir plusieurs résultats. Toutefois, et nous le verrons lors de l'implémentation, cela joue un rôle non négligeable sur les performances des algorithmes résultant de la transformation des axiomes.

GOGUEN et al. ('78) ont présenté une notation graphique claire et précise de la syntaxe d'un TDA. Nous présentons en vis-à-vis différents formalismes de spécifications syntaxiques, portant tous sur le même type, NATURAL, et ne comportant qu'un sous-ensemble utile des fonctions caractérisant le type.

Il est à remarquer que la démarche de spécification des TDAs permet la construction incrémentale par 2 mécanismes :

- La *restriction*, par suppressions d'opérations redondantes, qui résulte, dans la plupart des cas, de la démarche "Bottom-Up" (méthode ascendante) : on tente de spécifier le plus largement possible l'ensemble des opérations, puis on affine cette spécification par suppression d'opérations redondantes, et,
- L'*extension*, par adjonction de nouvelles opérations, entraînée par la démarche descendante ("Top-Down"). C'est la construction d'une spécification à partir d'une autre plus simple (réutilisation des connaissances), par adjonction de nouvelles opérations.

Ces mécanismes enrichissent le pool des opérations des structures de données, afin de prévoir l'avenir ou de l'affiner. Cela facilite grandement la maintenance et l'étape de spécification qui est traitée, de cette façon, en bloc.

c. Construction de la syntaxe d'une opération

Chaque opération caractérisant un type abstrait est une fonction, c'est-à-dire une relation mathématique entre certains arguments ayant une résultat intrinsèque. L'opération fournit un et un seul résultat, à partir d'un ensemble fini d'arguments. Nous écrirons indifféremment fonction ou opération, en signifiant les actions de transformation des types de données abstraits. Nous décrirons l'interface utilisateur de chaque opération, dont la présentation générale est la suivante :

```
Fonction nom_de_fonction ( liste_arguments ) donne resultat .

Soit  argument1   :  type_arg1 ,
        ...              ...
      argumentN   :  type_argN ,
      resultat    :  type_res.

Cette fonction donne un élément resultat de type type_res
qui est . . .     ( but de la fonction reprenant les arguments )

Préconditions : . . .

Postconditions : . . .
```

Le point 1 reprend le nom de l'opération, ainsi que les arguments dont l'ordre fait partie des spécifications à respecter par l'utilisateur de cette opération. Le résultat de la fonction ainsi définie est également énuméré. De plus, si la procédure ne doit pas être exportée à l'utilisateur, c'est-à-dire qu'elle sert de procédure de travail interne au module, alors on précise en tête le mot réservé : *private*.

La deuxième ligne permet de déterminer le type de chaque argument et du résultat, afin de simplifier le point 1 qui aurait pu être écrit (et ceci pour la clarté de la spécification) :

fonction nom_de_fonction (arg1 : type_arg1,...,argN : type_argN)
donne res : type_res .

L'explication informelle de "ce que fait" la fonction est décrite au point 3. Ce rôle débute, afin de standardiser la syntaxe, par :

"Cette fonction donne un élément de type ... qui est ..."

Un des avantages de cette notation est qu'elle ignore le style de programmation que l'on emploiera pour coder ces spécifications (généralité). En effet, alors que la notion de fonction existe dans les langages Pascal-like dont la syntaxe est très proche de la programmation logique, cette dernière manipule des fonctions dont l'exécution sera évaluée à vrai si la procédure a pu être menée à terme, à faux sinon. Le résultat de la fonction selon notre syntaxe est, dès lors, produit par *effet de bord* contrôlé, et subit le sort des arguments.

Enfin, le point 4 explicite les préconditions et les postconditions portant sur les arguments, car toutes les fonctions ne portent pas sur tous les éléments de l'ensemble. Ce sont bien souvent des fonctions partielles. Libre choix est laissé à l'utilisateur du module, en tant que programmeur, du respect de ces pré- et postconditions. Il pourra les vérifier dans le module appelant, ou les insérer dans le module dont ils font partie.

Il est bon d'insister sur le fait que ces conditions seront aussi précises que possibles, sans développement inutile alourdissant l'interface. De préférence, la même notation que celle utilisée pour la rédaction des axiomes équationnels sera choisie.

3. Spécification de la sémantique d'un TDA

Puisque nous savons que le type de données abstrait est défini par les fonctions le caractérisant de façon univoque (si l'ensemble de ces fonctions est cohérent), il reste à décrire le "comment faire" de chacunes d'elles.

Les fonctions seront décrites, selon la technique des types abstraits, par un ensemble fini d'axiomes. Ils seront relationnels, équationnels, ou même logiques. Nous nous sommes restreints aux axiomes de types équationnels.

a. Le langage de description axiomatique

La technique des types abstraits, se bornant à la structuration des objets au sein d'un même set (ensemble), ne déclare pas de langage axiomatique. Celui-ci devra être sélectionné selon l'option précisée plus haut. Dans le cadre de notre travail, et pour certaines raisons expliquées au chapitre suivant, c'est le langage des prédicats du premier ordre qui a retenu notre attention pour la rédaction des axiomes équationnels.

b. La démarche de conception des axiomes

Il est utile, dès ce moment, de parcourir la librairie de spécifications afin de déterminer ce qui a déjà été spécifié et qui peut être réutilisé, avec ou sans adaptation. Un moyen rapide d'y parvenir, mais qui ne pourrait être automatisé que partiellement de par la présence d'éléments informels, est la comparaison des spécifications syntaxiques des opérations.

Le mécanisme de construction des axiomes que nous promulguons ici est la RECURSION. D'autres existent, telle l'énumération par exemple.

C. Conclusion

Un formalisme aussi rigoureux que le langage des types de données abstraits, puisant ses sources dans les mathématiques, est idéal pour la représentation d'objets, sans prise en compte de détails de représentation interne.

Seule la syntaxe a été suffisamment décrite dans cette partie. Il reste à expliciter la sémantique des axiomes. Ce rôle est tenu par le deuxième chapitre qui expose un langage de description de ces axiomes : le langage des prédicats du premier ordre.

CHAPITRE 2 : Le langage des prédicats du premier ordre (FOPL)

A. Introduction

La logique des prédicats du premier ordre, F.O.P.L. (First Order Predicats Logic) permet, d'une part, de spécifier les axiomes des opérations caractérisant un type de données abstraits, et d'autre part, de rapprocher les spécifications des procédures exécutables.

La raison en est la double sémantique du langage, à la fois déclarative et procédurale.

Le langage FOPL présente, comme tout autre langage, une *syntaxe*, constituée des formules bien formées admises par la grammaire et, nous venons de le dire, une *sémantique* complète.

Nous décrirons dans un premier temps la syntaxe FOPL, nécessaire à l'écriture d'axiomes équationnels, avant d'envisager la sémantique de ce langage.

B. La syntaxe des prédicats du premier ordre

Une théorie du premier ordre est constituée de quatre parties :

- l'alphabet, c'est-à-dire l'ensemble des symboles unitaires,
- le langage de premier ordre, composé de l'ensemble des formules bien formées,
- l'ensemble des axiomes constructibles, et finalement,
- l'ensemble des règles d'inférence, axiomes universels.

1. L'alphabet FOPL

La table de la page suivante reprend l'ensemble des symboles permis pour la construction de phrases FOPL :

Symboles de

Variables	: [A-Z][a-zA-Z0-9]*
Constantes	: [a-z0-9spécial]+
Fonctions n-aires	: [a-zA-Z0-9spécial]+
Prédicats n-aires	: [a-zA-Z0-9spécial]+
Connecteurs	: [¬,&,∨, ⇐,⇒,⇔]
Quantificateurs	: [∀,∃]
Ponctuateurs	: ['(',')','{','}','[',']',',',]
Spéciaux	: [+,-,*,/,=]

- Les blancs soulignés sont permis partout dans un nom
- [symboles]+ signifie une suite de 1 à n symboles
  [symboles]* 0 à n symboles

Parmi ces symboles de base, seuls les trois derniers (connecteurs, quantificateurs et ponctuateurs) sont invariables, quelque soit l'alphabet décrit. De plus, seuls les symboles de constantes et fonctions peuvent être inexistants.

Remarque :

a) Selon l'usage courant, les symboles de fonctions sont appelés foncteurs.

b) L'arité d'une fonction est le nombre d'arguments que la fonction manipule, excepté le résultat qui est implicite à la fonction.

Afin d'assouplir l'écriture des formules, la logique des prédicats du premier ordre accepte la suppression des parenthèses non ambigües. A cette fin, les symboles fixes ont reçu une priorité de précédence. En voici la liste, de la plus importante à la plus faible, de haut en bas et de gauche à droite :

$$\neg,\ \forall,\exists,$$
$$\&,$$
$$\vee,$$
$$\Rightarrow,\Leftarrow,\Leftrightarrow.$$

Remarque : dans le cas de l'implication logique ($\Leftarrow$), nous dirons que le terme de droite est l'antécédent de l'implication, et celui de gauche, le conséquent.

Afin d'alléger l'écriture et la lecture des définitions qui vont suivre, nous prendrons f pour symbole de fonction et p pour symbole de prédicat.

2. Le langage des prédicats du premier ordre

Le langage proprement dit des prédicats du premier ordre est l'ensemble de toutes les formules bien formées construites avec les symboles permis de l'alphabet.

Un TERME (t) est
- une variable,
- une constante,
- une fonction dont les arguments sont des termes (récursivité de la définition) : $f(t_1, \ldots, t_n)$.

Une FORMULE BIEN FORMEE (F ou G) est

- un ATOME, (ou formule atomique) $p(t_1, \ldots, t_n)$,
- $(\neg F)$, $(F \,\&\, G)$, $(F \vee G)$, $(F \Rightarrow G)$, $(G \Leftarrow F)$, $(F \Leftrightarrow G)$,
- $(\forall x\ F)$, $(\exists x\ F)$.

L'ETENDUE (portée) d'un quantificateur limite l'occurence de la variable qui le suit immédiatement dans la formule. Quant aux autres variables, elles sont qualifiées de LIBRES.

- $(\forall x\ F)$: l'étendue de $\forall x$ est F
- $(\exists x\ F)$: $\exists x$

La FERMETURE d'une formule est l'extension à toutes les variables libres d'un quantificateur (soit universel, soit existentiel). De ce fait, une formule est fermée si aucune variable n'est libre.

Si un atome est positif dans F, il est:

- POSITIF dans $(F \,\&\, G)$, $(F \vee G)$, $(F \Rightarrow G)$, $(\forall x\ F)$, $(\exists x\ F)$.
- NEGATIF dans $(G \Leftarrow F)$, $(\neg F)$.

Un LITTERAL (L) est un atome (F) ou la négation de ce atome $(\neg F)$.

Une CLAUSE est une formule de la forme

$$\forall x_1 \ldots \forall x_s \, (L_1 \vee \ldots \vee L_m),$$

où x_i sont les variables apparaissant dans les littéraux Li.
Selon la définition d'un littéral, une clause peut être écrite de plusieurs façons équivalentes :

a) $\forall x_1 \ldots \forall x_s \quad (F_1 \vee \ldots \vee F_k \vee (\neg G_1) \vee \ldots \vee (\neg G_2))$,
b) $\forall x_1 \ldots \forall x_s \quad ((F_1 \vee \ldots \vee F_k) \Leftarrow (G_1 \,\&\, \ldots \,\&\, G_2))$,
c) $\qquad\qquad\quad (F_1 \vee \ldots \vee F_k \Leftarrow G_1 \,\&\, \ldots \,\&\, G_2)$,

Une NOTATION CLAUSALE SPECIALE, où toutes les variables sont supposées universellement quantifiées, sera choisie :

$$F_1, \ldots, F_k \Leftarrow G_1, \ldots, G_n$$

où ',' est le symbole séparateur de disjonction dans le conséquent,
de conjonction dans l'antécédent.

Une CLAUSE DE PROGRAMME est une clause dont le conséquent n'a qu'un atome, appelé tête de la clause.

$$F \Leftarrow G_1, \ldots, G_n$$

Une CLAUSE UNITAIRE est une clause de programme dont l'antécédent est vide.

$$F \Leftarrow$$

Un PROGRAMME est un ensemble fini de clauses de programme, et une PROCEDURE est l'ensemble de toutes les clauses dont le prédicat du conséquent est le même.

<u>exemple</u> : La procédure F est composée de m clauses :

$$F \Leftarrow G_{11}, \ldots, G_{1n}$$
$$\ldots \qquad\qquad \ldots$$
$$F \Leftarrow G_{m1}, \ldots, G_{mp}$$

Un OBJECTIF est une clause sans conséquent :

$$\Leftarrow G_1, \ldots, G_n$$

Une CLAUSE VIDE, ne comportant ni conséquent ni antécédent, est une contradiction.

Une CLAUSE DE HORN est soit une clause de programme, soit un objectif.

3. Les axiomes FOPL

Les axiomes sont des formules bien formées fermées. Ils sont utilisés pour dériver les théorèmes de la théorie.

4. Les règles d'inférence FOPL

Les règles d'inférence permettent de manipuler les axiomes. Celles-ci sont universelles, contrairement aux axiomes qui sont définis par le concepteur de la théorie du premier ordre. Dans le domaine de la preuve de théorèmes, la seule règle d'inférence utilisée est la *règle de résolution de Robinson,* que nous relatons plus en détail au chapitre 4, dans le cadre du langage Prolog.

C. La sémantique des prédicats du premier ordre

Il faut attacher à chaque symbole de chaque formule une signification précise, représentée par une valeur. Cette valeur est déterminée en logique mathématique par les valeurs de vérité "Vrai" et "Faux".

1. Sémantique des quantificateurs

$\forall$ universalité (pour tout) : assure la vérité de la formule à toutes les combinaisons de valeur de la variable concernée.

$\exists$ existentialité (il existe) : assure la vérité de la formule pour au moins une valeur de la variable concernée.

2. Sémantique des connecteurs

$\neg$	négation logique	$\Leftarrow$	implication logique
&	conjonction logique	$\Leftrightarrow$	équivalence logique
v	disjonction logique		

La sémantique des connecteurs est exprimée par la table de vérité ci-dessous :

F	G	$\neg$F	F&G	FvG	F$\Leftarrow$G	F$\Leftrightarrow$G
vrai	vrai	faux	vrai	vrai	vrai	vrai
vrai	faux	faux	faux	vrai	vrai	faux
faux	vrai	vrai	faux	vrai	faux	faux
faux	faux	vrai	faux	faux	vrai	vrai

3. la sémantique générale des FOPLs

Puisque les ponctuateurs sont des séparateurs, ils n'apportent aucune modification aux valeurs de vérité obtenues. Ainsi, les seules valeurs de vérité encore inconnues restent celles qui sont portées par les symboles de constantes, de fonctions et de prédicats. Ces valeurs dépendent d'une interprétation de la théorie FOPL.

Une INTERPRETATION définit un certain domaine de discours (ensemble non vide de valeurs) portant sur :

- l'étendue des variables (selon les quantificateurs),
- l'affectation à chaque symbole des éléments suivants :
  - . une constante d'un élément du domaine,
  - . une fonction d'une correspondance du domaine vers le domaine,
  - . un prédicat d'une correspondance du domaine vers une valeur de vérité.

L'interprétation spécifie les significations pour chaque symbole de la formule. De plus, pour une formule donnée, il peut y avoir plusieurs interprétations possibles, chacune exprimant une des deux valeurs de vérité, vrai ou faux.

Une PRE-INTERPRETATION (**J**) d'un langage (**L**) du premier ordre est l'union des éléments suivants :

- l'ensemble non vide **D** de valeurs (appelé le domaine),
- l'affectation à chaque constante de L d'un élément de D, et,
- l'affectation à chaque foncteur n-aire de L d'une correspondance de D_n vers D.

Une INTERPRETATION (**I**) est une PRE-INTERPRETATION (J) avec, pour chaque symbole de prédicat n-aire de L, une affectation d'une correspondance de D_n dans l'ensemble des valeurs de vérité {vrai,faux}. On dira, dès lors, que l'interprétation I est basée sur la pré-interprétation J.

Une AFFECTATION DE VARIABLE (**V**), selon la pré-interprétation J, est une affectation de chaque variable du langage de premier ordre L à un élément du domaine D de J.

Une AFFECTATION DE TERME, selon la pré-interprétation J et l'affectation de variable V, est composée comme suit :

- chaque variable reçoit son affectation selon V,
- chaque constante reçoit son affectation selon J,
- si $t'_1, \ldots, t'_n$ est l'affectation de terme de $t'_1, \ldots, t'_n$, et f' est l'affectation de terme de f, alors $f'(t'_1, \ldots, t'_n) \in D$ est l'affectation de terme de $f(t_1, \ldots, t_n)$.

Une formule du langage de premier ordre L peut recevoir une valeur vrai ou faux selon que F soit :

- $p(t_1, \ldots, t_n)$. Il faut calculer $p'(t'_1, \ldots, t'_n)$.
- $(G_1 \; ? \; G_2)$, où ? est un connecteur quelconque. Dans ce cas, on utilise la table de vérité du connecteur.
- $\exists x \; G$: F est vrai si $\exists d \in D$ tel que G soit vrai selon I et $\forall(x/d)$, faux sinon.
- $\forall x \; G$: F est vrai si $\forall d \in D$ tel que G soit vrai selon I et $\forall(x/d)$, faux sinon.

où $\forall$**(x/d)** signifie que la variable x est affectée à l'élément du domaine D.

En réalité, la valeur de vérité d'une formule fermée ne dépend pas de l'affectation des variables, ce qui permet de parler de la valeur de vérité d'une formule fermée selon une interprétation.

On dit d'une interprétation de la formule qui exprime la valeur vrai qu'elle est un MODELE de cette formule.

L'interprétation I est un modèle pour la formule bien-formée F si F est vrai selon I

Une formule F est dite SATISFAISANTE (respectivement non satisfaisante) dans l'interprétation I, si il existe au moins une affectation de variable qui donne la valeur vrai (respectivement faux) à F selon I.

Une formule F est dite VALIDE (resp. non valide) dans l'interprétation I, si toutes les affectations de variable donnent la valeur vrai, (resp. faux) à F selon I.

Une interprétation pour le langage du premier ordre L qui est un modèle pour chaque axiome de la théorie du premier ordre T est un modèle pour T. C'est ce qu'on nomme la T-EXTENSION.

Une interprétation pour le langage du premier ordre L qui est un modèle pour chaque formule d'un ensemble S de formules fermées est un modèle pour S. Il s'agit de la S-EXTENSION.

Une formule F est une CONSEQUENCE LOGIQUE (|=) de l'ensemble S des formules bien-formées fermées si, quelle que soit l'interprétation I du langage du premier ordre L, l'affirmation "I est un modèle pour S" implique que I est un modèle pour F.

Soit $S = \{F_1, \ldots, F_n\}$,

Alors $S \models F$ ssi $F_1 \,\&\, \ldots \,\&\, F_n \Rightarrow F$ est valide.

$S \models F$ ssi $S \cup \{\neg F\}$ est non satisfaisant.

Les FOPLs fournissent une méthode permettant de déduire les théorèmes d'une théorie, c'est-à-dire les formules qui sont des conséquences logiques des axiomes de la théorie. Ces formules sont vraies dans n'importe quelle interprétation qui est un modèle de chacun des axiomes de la théorie.

D'un point de vue programmation, la conséquence logique n'a pas un intérêt suffisant. L'utilisateur désire également connaître les liaisons faites pour les variables quantifiées existentiellement. C'est le mécanisme d'instanciation, amplement développé au chapitre 4, traitant de la programmation logique.

4. Restriction de la sémantique des FOPLs aux modèles de Herbrand

Le problème de base de la programmation logique est de déterminer, grâce au principe de résolution de Robinson, l'insatisfaisabilité d'une formule dans un ensemble S de formules bien formées fermées. Autrement dit, il faut montrer qu'aucune interprétation de $P \cup \{\neg G\}$ n'est un modèle.

Au regard de l'énormité de ce problème, de par la diversité des interprétations, il est possible de limiter l'interprétation à une interprétation de Herbrand.

L'INTERPRETATION DE HERBRAND est basée sur un domaine limité à l'ensemble de

- tous les termes de base formés à partir
  - . des constantes et
  - . des foncteurs, et de
- toutes les formules de base formées à partir
  - . des symboles de prédicat et
  - . des termes de base.

apparaissant dans le langage L du premier ordre.

Afin de visualiser clairement ce que chaque définition à suivre signifie, prenons un exemple repris de Lloyd (85) :

Soit P : $p(x) \Leftarrow q(f(x),g(x))$
$r(y) \Leftarrow$
Avec les symboles de prédicat p,q,r
fonction f,g
variable x,y

Un TERME DE BASE est un terme sans variable. De même, une FORMULE DE BASE est une formule sans variable.

Ce que l'on appelle UNIVERS DE HERBRAND (UI) est l'ensemble de tous les termes de base qui peuvent être formés à partir des constantes et des foncteurs apparaissant dans le langage L du premier ordre. Si L n'a pas de constantes, on en ajoute une (ici, a est un ensemble de constantes).

exemple :

UI = a,f(a), g(a), f(f(a)), f(g(a)), g(f(a)), g(g(a)), ...}

Quant à l'ensemble de toutes les formules de base pouvant être formées à partir des symboles de prédicats avec les termes de l'univers de Herbrand comme arguments, il porte le nom de BASE DE HERBRAND (BI).

exemple : BI = { p(a), p(f(a)), p(g(a)), p(f(f(a))), ... ,
q(a,a), q(a,f(a)), q(f(a),a), q(f(a),f(a)), ... ,
r(a), r(f(a)), r(g(a)), r(f(f(a))) , ... }

De là, la préinterprétation de Herbrand a comme domaine, l'univers de Herbrand, et l'interprétation de Herbrand a, comme base, la préinterprétation et la base de Herbrand.

Puisque les affectations aux constantes et foncteurs sont fixes, il est possible d'identifier une interprétation de Herbrand sur un sous-ensemble de la base de Herbrand. Ce dernier est l'ensemble de toutes les formules de base vraies dans l'interprétation.

Il sera plus utile de se référer, par abus de langage, à une interprétation d'un ensemble S de formules plutôt qu'au langage L sous-jacent duquel proviennent les formules. En particulier, l'ensemble S s'identifiera à l'ensemble P des clauses de programme.

5. Sémantique des FOPLs multi-types

Jusqu'à présent, on ne considérait l'appartenance de l'ensemble des variables et constantes qu'à un seul type. Or, il est courant que les axiomes manipulent des variables ayant des types différents. Selon la théorie multi-type, chaque affectation de variable devra être limitée à un type déterminé par le domaine de la variable.

exemple :

$\forall x\ F$ est remplacé par $\forall(\tau)x\ F$, ou $\forall x/\tau\ F$, qui signifie que la variable x est du type τ, c'est-à-dire que son affectation devra être faite dans le domaine de τ.

Nous n'allons pas aller au delà de ces explications, ce qui précède étant nettement suffisant pour la suite du travail.

D. Expression des axiomes TDAs en logique des prédicats premier ordre.

Le temps est venu de décrire formellement les éléments syntaxiques nécessaires à l'écriture des axiomes spécifiants les fonctions abstraites.

1. Définitions des éléments du langage (alphabet et formules)

a. Définitions d'ensembles

Les ensembles s'écrivent entièrement en majuscules.

VAR : ensemble des variables. Elles doivent commencer par une majuscule. La valeur qu'elles portent est déterminée à l'exécution des procédures implémentant les axiomes. Elles sont donc génériques.

CONST: ensemble des constantes. Ces éléments ont une valeur intrinsèque et inchangeable.

FONCT-N : ensemble des foncteurs (nom de fonctions) ayant n arguments. Ils s'écrivent entièrement en minuscules.

FONCT-0 : ensemble des foncteurs n'ayant aucun argument (liste des arguments vide). C'est donc un sous-ensemble de FONCT.

PRED: ensemble des prédicats. Précisons que nous utilisons un sous-ensemble du langage des prédicats du premier ordre, puisque nous nous limitons à un seul prédicat. C'est celui de l'égalité sémantique ('==').

CONSTRUCT : ensemble des constructeurs de type de données abstrait. C'est un sous-ensemble de foncteurs caractérisant univoquement le TDA.

TERME: ensemble des termes. Un terme est soit :
- une variable,
- une constante,
- un foncteur n-aire,
- un foncteur 0-aire.

Remarque:

FONCT ≡ FONCT-N U FONCT-O
TERME ≡ VAR U CONST U FONCT
FONCT ⊃ CONSTRUCT

b. Définitions d'éléments d'ensembles

Chaque élément appartenant à un ensemble représente en réalité une classe d'éléments semblables, dont chaque occurence peut être identifiée par un index. De plus, une liste d'éléments de même classe est représentée par l'élément générique surmonté d'un trait.

Soit X,Y,Z ∈ VAR;
a,b,c ∈ CONST;
f,g,h ∈ FONCT-N si $f(X_1,..,X_n)$, $g(Y_1,...,Y_n)$, $h(Z_1,...,Z_n)$;
f,g,h ∈ FONCT-0 si f(), g(), h();
t,u ∈ TERME;
cons ∈ CONSTRUCT.

2. Définition d'un sous-ensemble restreint d'axiomes FOPLs

Les foncteurs o-aires, ayant un domaine vide, sont toujours des constructeurs de base, puisqu'ils associent directement leur représentation syntaxique avec un élément appartenant à l'ensemble caractérisé par le type. Ainsi, ils n'ont pas à être définis sémantiquement par un ou plusieurs axiomes.

exemple :

Soit le type NATURAL caractérisant l'ensemble des éléments naturels dont la valeur est plus grande ou égale à zéro.
La fonction o-aire ***zero()*** *représente l'élément naturel dont la v valeur est 0 et dont la représentation mathématique est '0'.*

Quant aux constructeurs de base n-aires, ils n'ont également pas à être spécifiés axiomatiquement, puisque l'élément qu'ils représentent est obtenu par la relation abstraite et générale de construction de l'ensemble des éléments de tel type.

Une opération f (fonction) sera définie à l'aide d'un ensemble d'axiomes dont chacun a la forme générale suivante :

$$\overline{\mathbf{f(cons) == t}} \quad \text{où cons} \in \text{CONSTRUCT},$$

avec cons(u) et u ∈ (VAR ∪ CONSTRUCT), sous-ensemble de TERME. C'est donc un terme dont le symbole de tête est un constructeur de base, et dont les arguments ne peuvent être que des variables ou des constructeurs de base.

Cette forme a une signification contraignante, par l'usage d'un prédicat unique "==". Celui-ci limite la sémantique de la formule à une seule relation, dont le sens est :

f(cons) "est spécifiée sémantiquement par" t
"est définie par"
"est sémantiquement équivalente à".

a. Restriction de cohérence sur t

Le terme t ne peut être que :

- une variable reprise dans u,
- une variable qui est utilisée plus d'une fois, sinon elle ne correspondrait à rien, pouvant ainsi prendre n'importe quelle valeur,
- une constante,
- un terme dont le symbole de tête est un foncteur défini selon une relation de hiérarchisation (voir point suivant).

b. Notion de hiérarchie des spécifications

La dernière restriction sur t permet d'éviter les définitions cycliques, en ordonnant les spécifications par une relation de hiérarchie. Ainsi, l'architecture de spécification aura la définition suivante :

NIVEAU 0 ≡ CONSTRUCT ∪ VAR
NIVEAU i ≡ { f_i | ∀ j, 0 ≤ j ≤ i : f_i==f_j}

La signification fonctionnelle de la relation '==' veut que *"le résultat de l'application de f sur ses arguments est identique au résultat de l'application de t"*, où le résultat est l'ensemble des objets résultant d'une action de création et/ou de modification.

C'est une relation de hiérarchie qui définit un premier niveau contenant l'ensemble des axiomes dont la signification n'est exprimée qu'en terme de constructeurs primordiaux. Les niveaux suivants contiennent les axiomes dont la partie droite contient des termes spécifiés dans les niveaux inférieurs.

exemple :

conc(app(E,S),empty()) == app(E,S) Niveau 0
conc(empty(),S) == S Niveau 0
(S est un élément de type Séquence,
exprimé par des constructeurs de base uniquement)
conc(app(E,S),S2) == app(E,conc(S,S2)) Niveau 1

Donc cons est un élément minimal, et CONSTRUCT l'ensemble des éléments minimaux. Ainsi, toute opération est spécifiée axiomatiquement en terme de constructeurs de types abstraits, *in fine*.

exemple :

si S2 est empty(), et S est app(E2,empty()),
alors conc(app(E,S),S2) == app(E,conc(S,empty())
== app(E,app(E2,empty()))

c. Modes de spécifications axiomatiques

Deux modes ou méthodes de spécifications axiomatiques des TDAs sont possibles :

- DECOMPOSITION de fonctions en fonctions plus simples (plus proches de CONSTRUCT), où une fonction est définie en terme d'autres selon la relation d'ordre '=='.

$$f == f' \text{ où } f' \equiv FONCT \setminus f$$

- RECURRENCE, où une fonction est définie en terme d'elle-même selon une relation d'ordre bien fondée sur les arguments.

$$f == f' \text{ où } f' \equiv f$$

avec au moins un argument plus petit selon une relation d'ordre bien fondé. Le choix de cette relation est dépendant d'un autre axiome spécifiant f, qui permet de définir f de manière constante, ou par décomposition (élément minimal).

d. Condition de complétude de la spécification de f

L'opération f sera spécifiée sémantiquement de façon complète (condition suffisante mais pas nécessaire) si toutes les combinaisons de constructeurs possibles selon le type de l'argument, pour chaque argument, sont reprises dans la spécification.

$$\begin{aligned} f(a_{11},\ldots,a_{n1}) &== t_1 \\ \ldots &== \ldots \\ f(a_{1m},\ldots,a_{nm}) &== t_m \end{aligned}$$

où n est le nombre d'arguments de f,
m est le nombre minimalement suffisant d'axiomes spécifiant f,
$\forall i, \forall j : a_{ij} \in$ CONSTRUCT.

La spécification de f sera complète si :

$$m = P_{i=1}^{n} k_i$$

avec k_i, le nombre de constructeurs de base du type abstrait auquel appartient l'argument i.

exemple :

length(S) n=1; k_1=2; donc m=2
conc(S_1,S_2) n=2; k_1=2; k_2=2; donc m=4

Cependant, cet ensemble complet d'axiomes spécifiant f peut présenter des redondances de spécification du sens d'une fonction qu'il convient d'éliminer par généralisation. Il faut comprendre "redondance" comme un ensemble d'axiomes à même foncteur de gauche, dont les parties droites sont sémantiquement identiques, et qui peuvent être généralisés en un seul axiome, à un jeu de substitution d'argument près.

exemple :

conc(empty(),empty()) == empty()
conc(empty(),app(E,S)) == app(E,S)
conc(app(E,S),empty()) == app(E,S)
conc(app(E,S),app(E_2,S_2)) == app(E,conc(S,app(E_2,S_2)))

devient

conc(empty(),S) == S
conc(app(E,S),S_2) == app(E,conc(S,S_2))

puisque

les deux premiers axiomes peuvent être définis par un seul axiome général, tout comme les deux suivants, par généralisation du second argument. Bien que cela soit évident pour la première paire, montrons la transformation de la seconde :

conc(app(E,S),empty()) == app(E,S) == app(E,conc(S,empty())
conc(app(E,S),app(E_2,S_2)) == app(E,conc(S,app(E_2,S_2)))

Cette élimination de la redondance impose qu'il soit possible d'utiliser des variables en lieu et place des constructeurs de base. Ces variables représentent les k_i constructeurs de type de l'argument.

Les axiomes prennent maintenant la forme générale suivante :

$$f(\overline{cons},X) == t \text{ où } X \in VAR.$$

La méthode de construction de spécifications axiomatiques par induction structurelle, présentée au chapitre suivant, est précisément intéressante parce qu'elle fournit des spécifications complètes non redondantes.

E. Conclusion

Le lecteur aura maintenant la certitude qu'il dispose d'un outil suffisant pour décrire un type de données abstrait. Les raisons de cette certitude sont la simplicité et la standardisation de la syntaxe des TDAs, l'utilisation d'un langage simple et restreint, et la possibilité d'emploi de méthodes de hiérarchisation.

Nous n'avons pas encore jugé utile d'expliciter la sémantique des TDAs en logique des prédicats du premier ordre. Elle sera précisé dans la partie traitant de l'étape de transformation des axiomes en procédures Prolog (chapitre 5).

Le chemin est encore long avant que l'on sache écrire les spécifications axiomatiques correctement, sans défauts tels que la redondance, l'incohérence, l'incomplétude. Ceci peut s'apprendre à l'usage, bien qu'il serait plus utile que l'on dispose d'une méthode de construction d'axiomes complets et cohérents. Ceci est le sujet de notre prochain chapitre.

CHAPITRE 3 : Conception de spécifications axiomatiques équationnelles correctes

L'analyste-spécifieur se trouve confronté au problème courant de l'exactitude des spécifications par rapport à elles-même et à la réalité du problème. Bien des ouvrages traitent de ce fossé et peu de solutions valables et complètes ont été proposés pour le réduire.

Cette partie expose une méthode simple de construction de la sémantique des TDAs, en parallèle avec la méthode résultant des travaux de DEVILLE (87) dans le domaine de la construction d'algorithmes logiques par induction structurelle. Notre méthode permet d'obtenir des axiomes équationnels décrivant les fonctions d'un type de données abstrait quelconque, tout en étant certain de l'exactitude de ceux-ci. Elle applique la technique de spécification en types de données abstraits à partir de la description informelle.

A. Introduction et prérequis

La réalité peut être exprimée de plusieurs manières, que ce soit dans le langage courant, dans un sous-ensemble de ce langage sur lequel on impose des contraintes renforçant le contrôle de cohérence et de validité, ou dans certains langages possédant des puissances d'expression accrues tout en s'éloignant de la perception rapide et simple de ce qu'ils veulent dire.

En général, la puissance de l'explicitation mathématique est acquise en considérant des structures infinies ou potentiellement infinies. Les instances finies n'en sont qu'un cas particulier limitant les possibilités.

En informatique, les programmes offrent cette "potentialité", qualificatif judicieux, par la méthode d'appels récursifs. De même, une façon simple de spécifier une opération dont le domaine d'analyse peut être infini est d'utiliser la récursivité. Heureusement, le langage des types de données abstraits, couplé au langage des prédicats du premier ordre, permet ce type de mécanisme.

Dans ce chapitre, nous ne nous intéresserons qu'à la construction d'axiomes équationnels dont le prédicat de base est l'égalité sémantique ('=='), par le biais de la récursivité. En effet, une autre façon de réaliser une opération est de l'exprimer par décomposition. Ceci ne présente aucun problème pour le lecteur, soyons-en certain. De plus, aucune structure proposée par les langages classiques telles le test ou l'itération ne sont possibles en spécification de TDAs.

Rappelons, à bon escient, la signification de quelques symboles importants :

== : égalité sémantique (le terme de gauche a la même signification que le terme de droite, bien qu'ils soient différents syntaxiquement).

= : égalité syntaxique (le terme de gauche doit être identique, à un jeu d'instanciation près, au terme de droite).

⇔ : équivalence logique (Si l'un est vrai, alors l'autre aussi, sinon l'autre est faux, et vice-versa).

⇐ : implication logique (Si le prédicat de droite est vrai, alors celui de gauche l'est aussi).

B. Critiques de spécifications sémantiques textuelles

Les éléments en entrée de la méthode d'induction structurelle sont les spécifications sémantiques textuelles (et, de ce fait, informelles), que nous voudrions formaliser. Cette explicitation du rôle d'une fonction doit éviter, comme tout texte de spécification, les sept péchés, tels que l'ambiguïté ou la surspécification, ...

Dans ce cadre, présentons quelques exemples de spécifications sémantiques textuelles, et critiquons les.

1. Exemple A : Suppression d'un élément d'une liste

> *"Le prédicat efface(X,Y,Z) enlève la première occurence de l'élément X de la liste Y, fournissant une nouvelle liste Z, supprimée. Si X n'est pas dans la liste, le prédicat échoue."*

Il est impossible de connaître quel sont les paramètres devant être instanciés avant l'exécution de la procédure. La procédure Prolog correspondante sera exacte par rapport aux spécifications si la directionnalité est :

in(ground,ground,var) : out(ground,ground,ground).

Tout autre emploi de la procédure mènera à des résultats incorrects. Ceci montre qu'une description de la directionnalité dans une spécification donne des informations indispensables pour l'utilisateur (Deville '87).

Dans notre cas, la description de la directionnalité sera inutile, les types de données abstraits ayant une directionnalité implicite de par la notion mathématique de fonction. De plus, la seconde partie du mémoire démontre que notre objectif reste une programmation fonctionnelle, malgré le choix de la programmation logique comme outil d'implémentation.

2. Exemple B : Evaluation d'une expression

"La procédure 'X est Y' réussit si X équivaut à l'évaluation de l'expression Y. Le terme n'a pas d'effet de bord et ne peut pas être resatisfait. Une erreur d'exécution est produite si X n'est pas une variable non instanciée ou un nombre, ou si Y n'est pas une expression arithmétique".

Cet exemple décrit des préconditions sur les types, la directionnalité et également les effets de bords. Toutes les informations nécessaires pour l'utilisation correcte de la procédure sont présentes. Toutefois, rien n'est structuré, et peut-être trop opérationnel.

En conclusion, exprimer la signification d'une procédure (fonction dans notre cas) en français n'est pas simple, et demande d'être rigoureux. A cette fin, la description du rôle d'une fonction doit être claire, simple et précise. Trois critères sont à retenir :

- la directionnalité (implicite ici),
- les préconditions et postconditions,
- les effets de bords éventuels.

Pour ce dernier critère, précisons que les types de données abtraits spécifient des fonctions dont le domaine doit être invariable, alors que le codomaine est la partie modifiée ou crée. Ainsi, les effets de bords sont éliminés par convention et restriction sur la syntaxe des fonctions.

C. La construction d'axiomes équationnels par induction structurelle

Deville ('87) décrit, dans son troisième chapitre, les algorithmes logiques, dont il présente la construction d'une première version par une approche de l'induction structurelle. De plus, il démontre l'exactitude de tels algorithmes. La partie à venir s'inspire largement de ces travaux. Après une présentation du principe de l'induction structurelle, nous développerons la théorie de notre méthode. Enfin, nous décrirons les différences avec la méthode de construction des algorithmes logiques.

1. Principe de la construction par induction structurelle

L'idée de base de la *construction par induction structurelle* est de déterminer la signification d'un terme, en considérant certaines structures différentes possibles de ses arguments, représentatives d'un ensemble de formes structurelles. Selon ces structures, on pourra construire le terme à droite du signe d'égalité ou d'équivalence.

exemple :

Si l'on veut déterminer une opération de concaténation entre deux séquences, on déterminera le résultat selon les formes possibles des séquences arguments : la séquence vide et la séquence d'au moins un élément.

Quelques concepts formels sont nécessaires pour définir le principe de *l'induction structurelle.*

Déf. Une *séquence descendante* d'éléments d'un ensemble quelconque E est une séquence établie par une relation d'ordre '>' telle que :

$$x_n > \ldots > x_{i+1} > x_i > x_{i-1} > \ldots > x_1$$

Déf. Un ensemble E ordonné par une relation '>' est un *ensemble bien fondé* si et seulement si il n'y a pas de séquence infinie décroissante des éléments de E selon la relation '>'.

Déf. L'élément e est *l'élément minimal* de l'ensemble E selon la relation '>' si et seulement si

$$e' < e \text{ n'existe pas, } \forall e' \in E$$

Déf. Un ensemble E ordonné par une relation '>' est un *ensemble bien fondé* si et seulement si tout sous-ensemble non vide de E a un élément minimal selon '>'.

Cette dernière définition est déduite des deux précédentes. Maintenant, il est possible de définir le principe général d'induction.

Principe général de l'induction.

Soit (E,<) un ensemble bien fondé,
I un sous-ensemble de E
Si I contient n'importe quel élément e $\in$ E chaque fois qu'il contient tous les éléments e' $\in$ E tel que e' < e,
alors I = E

L'induction structurelle est une induction selon une relation d'ordre bien fondé qui porte sur la structure des éléments de l'ensemble E.

Application du principe d'induction

Une façon d'utiliser le principe d'induction est de suivre les étapes suivantes, dont l'objectif est de prouver qu'une propriété W est vraie.

- Définir une relation bien fondée > sur l'ensemble E,
- Prouver que $W(x_0)$ est vrai pour tout élément minimal x_0 appartenant à E,
- Pour tout x non minimal, prouver que W(x) est vrai en assumant que W(y) est vrai pour y < x,
- Finalement conclure que W(x) est vrai pour tout élément de E.

L'aspect créatif probablement le plus important de l'application du principe d'induction est la détermination d'une relation bien fondée idéale.

2. La méthode de construction d'axiomes équationnels

La présentation suivante est celle d'une méthode de construction d'axiomes équationnels écrits en langage de la logique des prédicats du premier ordre. Nous affirmons que les axiomes obtenus sont corrects par construction. Notre propos n'est pas de le démontrer, ceci pouvant faire l'objet d'un autre mémoire.

a. Définition de la sémantique d'une fonction

Le chapitre deuxième explique qu'une fonction peut être définie sémantiquement par un ensemble d'axiomes équationnels. Chaque axiome précise, en fait, quelle doit être la suite des fonctions à effectuer pour obtenir le résultat de la fonction à définir, pour des occurences particulières de chaque paramètre.

La raison est qu'une fonction donne un résultat typé, mais variable. Il sera différent selon l'instanciation des arguments de la fonction.

$f(x_1,\ldots,x_k)$ est définie par F_1 ou F_2 ou ... ou F_k,

où F_i définit la suite des opérations à effectuer pour obtenir un résultat d'une certaine valeur, dans le cas où certains paramètres x_i sont instanciés à certaines valeurs v_j

b. Définition des axiomes équationnels

Définition générale d'un Axiome équationnel

$$f(t_1,\ldots,t_n) == t_{def}$$

où $f \in$ FONCTEUR,

$t_1,\ldots,t_n \in$ VAR $\cup$ CONST,

$t_{def} \in$ TERME,

$n \geq 1$

Le seul prédicat présent d'un axiome équationnel est l'égalité sémantique entre deux termes. En réalité, nous nous intéressons à la relation d'équivalence sémantique entre deux foncteurs. Le premier foncteur n'a, comme arguments, que des termes du type variable ou constante. Notre idée est de définir le sens d'une fonction qui manipule les constructeurs de base d'un type de donnée abstrait.

Le terme de droite est une composition de fonctions. Celles de plus bas niveau calculent la valeur des arguments de la fonction qui les contient. Ce terme est défini par deux règles d'égalité syntaxique, dont une est récursive :

a) t_{def} **= cons, avec cons** $\in$ **CONSTRUCT**

b) $t_{def} = f(t_1,...,t_k)$,
avec $t_1,...,t_k$ **eux-mêmes définis comme** t_{def}

exemple : Les compositions de termes suivants sont des formules valides :

add(succ(zero()),pred(soust(zero(),succ(zero()))))
conc(app(E,empty()),app(E2,app(E,app(E3,empty()))))

L'évaluation du résultat de ces expressions se fait en profondeur, afin de ramener l'expression a une composition de constructeurs de base des TDAs.

Ainsi, nous ne voulons pas définir des formules permettant de vérifier que le prédicat '==' est vrai, mais, en affirmant que ce prédicat est vrai, il faut déterminer quel est le terme t_{def} qui est le résultat de l'application de la fonction f à ses arguments $t_1,...,t_k$

La sémantique de la forme axiomatique précédente est :

"*L'application de la fonction f sur ses arguments donne le même résultat que l'évaluation du terme* t_{def} ".

c. Le principe général de la méthode de construction d'axiomes par induction structurelle

L'idée sous-jacente à la méthode est de reprendre le principe de l'induction pour permettre la détermination des formules t_{def}. Comme ces formules sont dépendantes de l'instanciation particulière des arguments de la fonction à spécifier, il faudra faire une induction des formules sur base des différentes instances de chaque argument.

Ceci implique, puisque nous manipulons des fonctions caractérisant des types de données abstraits, que l'on devra déterminer ces formules pour chaque combinaison des constructeurs de base des types des arguments.

La démarche d'induction sera incrémentale : d'abord choisir un paramètre d'induction. Si, pour les différents constructeurs de base de ce paramètre, il est possible de définir une formule, alors l'objectif est atteint. Sinon une nouvelle analyse sur un autre paramètre d'induction sera faite.

Traçons brièvement les étapes successives nécessaires à l'obtention de spécifications sémantiques de la fonction $f(t_1,...,t_k)$, étapes que nous détaillerons plus loin exemples à l'appui :

1. Choisir un ou plusieurs paramètres t_j parmi ceux de la fonction à spécifier $(t_1,...,t_k)$,

2. Définir une relation bien fondée sur le type des paramètres t_j

3. Déterminer les constructeurs de base couvrant toutes les formes structurelles différentes de t_j, et dont un au moins est une constante, ou fonction 0-aire ou n-aire sans rgument de la sorte d'intérêt (c'est à dire un élément minimal selon la relation d'ordre bien fondé sur le type du paramètre d'induction). Ceci assure la *complétude* de la spécification.

4. Construire les termes t_{def} correspondants à chaque k-tuple de constructeurs de base pour les arguments t_1 à t_k respectant les préconditions de la spécification de f. Ceci assure la *cohérence* de la spécification.

Remarque :

Rappelons que les constantes (ou valeurs des éléments de la sorte d'intérêt du type abstrait) sont déterminées par des compositions de constructeurs de base.

d. La réalisation pratique de la méthode

La méthode de construction d'axiomes équationnels est donc accomplie en quatre étapes. Elle assure que les axiomes seront corrects.

Le canevas général d'une construction d'axiomes équationnels par induction structurelle, permettant d'obtenir un terme de la forme précitée, est schématisé à la page suivante :

Paramètre d'induction : argument(s) choisi(s)

Relation bien-fondée : relation d'ordre sur les formes de(s) l'argument(s)

Forme structurelle du paramètre d'induction :

C_1: .....
⋮ ⋮
C_n: .....

Construction des F_i :

Si C_1 alors .....
Donc F_1 est .....
⋮
Si C_n alors .....
Donc F_n est .....

La recherche de chacun des ces quatre points est une partie de l'aspect créatif de la preuve par induction. L'objectif est d'obtenir un ensembles d'axiomes corrects par construction, en se limitant au modèle de Herbrand.

Paramètre d'induction

Le premier pas est le choix du ou des paramètres d'induction, c'est-à-dire d'un ou de plusieurs paramètres de la fonction à spécifier dont on déterminera les différentes formes structurelles et qui engendreront une définition de terme de la partie droite de l'équivalence sémantique.

Deux méthodes sont possibles selon que l'on choisit un argument (méthode ascendante) ou le résultat (méthode descendante) de la fonction à spécifier, comme paramètre.

Seule la méthode ascendante dans le choix du paramètre d'induction convient à notre méthode. Celui-ci doit être sélectionné parmi les arguments de la fonction f-fonction à spécifier. Les raisons sont les suivantes :

- Tout d'abord, dans notre objectif fonctionnel, le résultat est implicite à la fonction, il ne fait pas à proprement parler partie de la liste des arguments.

- Ensuite, selon une heuristique fonctionnelle et directionnelle, elle respecte mieux le sens de "résultat à déterminer".
- Enfin, il semble plus aisé de trouver une relation bien-fondée. Et cette relation reflètera la structure du paramètre d'induction.

Définissons les heuristiques possibles pour le choix des paramètres d'induction :

Déf. L'HEURISTIQUE FONCTIONNELLE porte son choix sur un P.I. tel que, donnant une instance close de celui-ci, la relation d'équivalence sémantique est valable pour au moins une instance close des autres paramètres.

Déf. L'HEURISTIQUE DIRECTIONNELLE porte son choix sur un P.I. qui est toujours clos dans la partie input de la directionnalité spécifiée.

Lorsque deux ou plusieurs candidats sont possibles, et que rien ne permet d'affirmer que l'un facilitera plus le traitement, il ne reste qu'à choisir arbitrairement un de ceux-ci, ou même les deux.

exemple (Cet exemple servira pour l'illustration des autres points)

Soit la fonction conc qui fournit la séquence résultant de la concaténation des deux paramètres de type séquence .

conc(Seq_1,Seq_2) --> Seq_res

Choisissons arbitrairement le premier paramètre comme paramètre d'induction : c'est une séquence (Seq_1)

Relation bien-fondée

La relation d'ordre bien-fondé doit assurer qu'il n'existe pas de séquence infinie décroissante des diverses formes du paramètre d'induction, et donc qu'il existe au moins une forme structurelle minimale dans cette séquence selon la relation.

Puisque les formes structurelles du paramètre d'induction portent sur les constructeurs de base, il sera donc indispensable de posséder au moins un constructeur 0-aire ou n-aire sans arguments de la sorte d'intérêt.

exemple :

P.I. : Seq_1 de type séquence.
R.B.F. : s1 < s2 si s1 est un suffixe propre de s2.

Une relation d'ordre bien fondé existe sur les formes structurelles d'une séquence. En effet, une séquence peut être vide, ou avoir un élément de tête de type ELEM et une queue S qui est un suffixe propre de la séquence initiale. La relation doit assurer qu'il existe un élément minimal. Celui-ci est la séquence vide, qui est la plus petite séquence envisageable (le plus petit suffixe).

En conclusion, on choisira une relation bien-fondée en connaissance des constructeurs minimaux et des constructeurs récursifs (dont un paramètre au moins est du type à spécifier).

Forme structurelle du paramètre d'induction

Nous devons construire le terme t_{def} (composition de termes) pour toutes les combinaisons C_i des constructeurs de base des paramètres d'induction. Il est donc utile de reprendre, avant tout, les constructeurs de base de chaque paramètre et de déterminer les combinaisons possibles. Le choix des constructeurs possibles pour un paramètre découle, bien entendu, de la relation bien fondée.

C_1, C_2, ..., C_n doivent couvrir toutes les formes possibles du paramètre d'induction : pour n'importe quelle instanciation x du paramètre, nous avons $X=t_1$ ou $X=t_2$ ou ... ou $X=t_n$ (t_1,...,t_n sont les constructeurs de base possibles). C'est afin d'assurer la complétude de la spécification axiomatique.

exemple :

Constructeurs possibles de la séquence, selon la relation d'ordre bien fondé choisie et les préconditions assurant la bonne application de la fonction :

forme fixée empty : séquence vide
forme récursive app(E,S) : séquence d'au moins un élément (d'un élément E et d'une sous-séquence S)

Construction des Formules sémantiquement équivalentes à la fonction

Les formules F_i déterminant un résultat de la fonction à spécifier sont des compositions de termes t_{def}. Chaque Fi doit fournir le résultat qui doit être obtenu selon l'application de la fonction pour la combinaisons d'instances des paramètres satisfaisant les préconditions de la spécification.

Remarque :

Par souci de terminologie adéquate, nous parlerons de formules fonctionnelles.

Les formules peuvent être construits de deux manières possibles :

- Par *réduction de problèmes* , en réduisant le problème à de plus simples sous-problèmes, c'est-à-dire, en spécifiant le résultat par l'application d'autres fonctions déjà spécifiés, ou supposés telles.

- Par *emploi récursif* de la fonction sur des termes dont la structure est plus proches de l'élément minimal de la relation bien-fondée. Cela revient à dire que la nouvelle instance du paramètre sera soit un constructeur minimal, soit un constructeur dont les arguments de la sorte d'intérêt sont des constructeurs plus proches d'un constructeur minimal (récursivité de la définition).

exemple :

1. Réduction du problème :

Soit une fonction de suppression de tous les éléments E d'une séquence S. On peut dire que supprimer tous les éléments E de S revient à supprimer le premier élément de S, puis à supprimer tous les éléments de la suite résultante. (Il y a, à la fois, réduction et emploi récursif dans cet exemple).

```
del_all(E,empty()) == empty()
del_all(E,S) == del_all(E,del_first(E,S))

del_first(E,empty()) == empty()
del_first(E,app(E,S)) == S
del_first(E,app(E2,S)) == app(E2,del_first(E,S))
```

2. Emploi récursif de la fonction

- si Seq_1 est empty(),
  alors F_1 est Seq_2
  (La séquence concaténée est syntaxiquement égale à la seconde séquence).
- si Seq_1 est app(E,S) ,
  alors F_2 est app(E,conc(S,Seq_2)

conc(empty(),Seq_2) == Seq_2
conc(app(E,S),Seq_2) == app(E,conc(S,Seq_2)

3. Quelques exemples de construction d'axiomes

Exemple 1 : Détermination de la longueur d'une séquence

La fonction *length(S)* fournit un naturel N qui est égal au nombre d'éléments de la séquence S.

Choix du paramètre d'induction

Seul l'argument S peut et doit être retenu, puisqu'il est le seul argument de la fonction, et qu'il possède plusieurs formes structurelles possibles.

Relation d'ordre bien fondé

s1 < s2 ssi s1 est un suffixe propre de s2

Formes structurelles du P.I.

S : empty()
app(E,S1)

Construction des formules F_i

- Si la séquence est vide, elle n'a pas d'élément donc sa longueur est nulle. F_1 = zero()
- Si la séquence a au moins un élément et une sous-séquence, sa longueur est celle de la sous-séquence plus 1. (La sous-séquence est plus petite que la séquence qui la contient).
  F_2 = succ(length(S1))

Axiomes obtenus

length(empty()) == zero()
length(app(E,S1)) == succ(length(S1))

Exemple 2 : Détermination de l'élément à telle position dans une séquence

La fonction *ith(N,S)* fournit un élément E qui est le $N^{ième}$ de la séquence S (à la $N^{ième}$ position).

Choix du paramètre d'induction

N et S sont deux candidats possibles. Choisissons N, par convention.

Relation d'ordre bien fondé

n1 < n2 ssi n1 a une valeur plus petite que n2 dans l'ensemble mathématique des naturels (c'est-à-dire si n1 est plus proche de zéro que n2).

Formes structurelles du P.I.

N : zero()
succ(N1)

Construction des formules F_i

- Si la position est nulle, on ne peut pas y trouver d'éléments. Donc, il y a erreur lors de l'utilisation de la procédure exécutable avec cette valeur d'argument.
- Si la position est succ(N1), il faut faire une induction sur N1.
    - Si N1 est zero(),
        - l'élément est celui qui est en tête de la séquence si elle est app(E,S1). F_1 = E.
        - Si la séquence est vide, il y a erreur : mauvaise position (hors de la limite permise)
    - Si N1 est succ(N2),
        - l'élément est celui qui est en position N1 de la sous-séquence S1 de S, F_2 = ith(N1,S1)
        - Il y a erreur si la séquence S n'a pas de sous-séquence.

Axiomes obtenus

ith(succ(zero()),app(E,S1)) == E
ith(succ(N1),app(E,S1)) == ith(N1,S1)

Exemple 3 : Vérification de l'appartenance d'un élément à une séquence

La fonction *is_in(E,S)* fournit un booléen B qui est vrai si l'élément E est dans la séquence S, faux sinon.

Choix du paramètre d'induction

Seul l'argument S peut et doit être retenu, puisqu'il est le seul argument de la fonction dont on connait les différentes formes structurelles possibles. E est un élément dont le type est quelconque.

Relation d'ordre bien fondé

s1 < s2 ssi s1 est un suffixe propre de s2

Formes structurelles du P.I.

S : empty()
app(E1,S1)

Construction des formules F_i

- Si la séquence est vide, elle n'a pas d'élément donc E ne peut appartenir à la séquence. F_1 = false()
- Si la séquence a au moins un élément et une sous-séquence, il faut vérifier que l'élément en tête de la séquence E1 est le même que E.
  - Si E1=E, alors E est dans la séquence S. F_2 = true()
  - Si E1≠E, alors E est peut-être dans la sous-séquence S1 de S. F_3 = is_in(E,S1)

Axiomes obtenus

is_in(E,empty()) == false()
is_in(E,app(E,S1)) == true()
is_in(E,app(E1,S1)) == is_in(E,S1)

Dans cet ensemble d'axiomes, l'ordre a de l'importance, puisque si l'on essaie le dernier axiome avant le deuxième, on oubliera de vérifier que E=E1.

Une autre façon d'exprimer les deux derniers axiomes, et plus formelle, est présentée à la page suivante:

is_in(E,app(E1,S1)) == or(equal(E,E1),is_in(E,S1))

Ici, cela signifie que soit E1 est syntaxiquement égal à E, soit E est dans S1. (ou logique).

Une dernière possibilité est d'utiliser les préconditions :

if E=E1 then is_in(E,app(E1,S1)) == true()
if not(E=E1) then is_in(E,app(E1,S1)) == is_in(E,S1)

4. Présentation succinte de la méthode de construction d'algorithmes logiques

La méthode de construction qu'a développé Deville ('87) permet d'obtenir rapidement, et avec la certitude de leur exactitude, des algorithmes logiques, et cela par le principe de l'induction structurelle.

L'objectif poursuivi est d'obtenir des algorithmes logiques corrects par construction, tout en ne considérant que le modèle de Herbrand, c'est-à-dire qu'il est possible de trouver des valeurs constantes pour chaque variable des paramètres telles que la formule est vraie selon cette interprétation. Les algorithmes logiques résultant sont sous une forme restreinte.

a. définition d'un algorithme logique

Définition générale d'un Algorithme Logique

$\forall X_1 \ldots \forall X_n \; (p(X_1 \ldots X_n) \Leftrightarrow F)$

où n >= 0, F est une formule.

Si F est de la forme : $(\exists Y_1 \ldots \exists Y_n)$

avec $X_i <> Y_i \quad \forall i,j$

alors, on réduit en $p(X_1 \ldots X_n) \Leftrightarrow$ Déf

Dérivons de cette formule bien-formée fermée, ou algorithme logique, une autre formule plus aisément manipulable :

$$p(x) \Leftrightarrow \begin{array}{l} C_1 \;\&\; F_1 \\ \vee\; C_2 \;\&\; F_2 \\ \quad\vdots \quad\;\; \vdots \\ \vee\; C_n \;\&\; F_n \end{array}$$

où $p(x)$ est un prédicat avec un ensemble x d'arguments,
C_i est une conjonction de littéraux,
F_i est une formule bien formée.

Les variables de F_i et C_i qui ne sont pas présentes dans p doivent être quantifiées existentiellement.

Décomposons la forme générale en plusieurs formules. Toutefois, il faut veiller à modifier l'équivalence en implication.

$$\begin{array}{l} p(x) \Leftarrow C_1 \;\&\; F_1 \\ p(x) \Leftarrow C_2 \;\&\; F_2 \\ \quad\vdots \quad\;\; \vdots \\ p(x) \Leftarrow C_n \;\&\; F_n \end{array}$$

b. Le principe général de la méthode de construction d'algorithmes logiques par induction structurelle

Les étapes successives nécessaires à l'obtention de spécifications sémantiques du prédicat logique p, sous forme d'algorithme logique AL(p), sont pratiquement identiques aux étapes décrites pour les axiomes équationnels, bien que des différences conceptuelles les distinguent :

1. et 2. Ces étapes de choix des paramètres d'induction et de définition d'une relation bien fondée sur le type de ces paramètres sont semblables. Toutefois, les paramètres sont ceux du prédicat p et non plus de la fonction f.

3. Construire les conjonctions de littéraux C_i de toutes les formes structurelles différentes de x_j(avec au moins une forme couvrant l'élément minimal de la relation bien fondée)

4. Construire les formules F_i correspondantes tel que $\exists (C_i \& F_i)$ est une condition nécessaire et suffisante afin d'avoir un n-tuple de base respectant les préconditions de la spécification de p et appartenant aux paramètres de p quand $\exists (C_i)$ est vrai.

c. La réalisation pratique de la méthode

La méthode de construction d'algorithmes logiques se base sur le même canevas général que celui d'une construction d'axiomes.

Paramètre d'induction

Le premier pas est le choix du ou des paramètres d'induction, c'est-à-dire d'un ou de plusieurs paramètres du prédicat à spécifier (correspondant à notre fonction dans lequel le résultat apparaît explicitement, sous forme d'un paramètre supplémentaire) qui seront décomposés dans leurs différentes formes structurelles.

Le choix ne se portera pas sur un terme, étant donné la difficulté de distinguer les différentes formes structurelles de celui-ci. En outre, une relation bien-fondée sur un terme ne simplifierait pas le problème sur les formes de ce terme.

exemple

Soit le prédicat p-conc qui est vrai si le troisième paramètre est la séquence résultant de la concaténation des deux autres paramètres de type séquence .

p-conc(Séq_1,Séq_2,Séq_res)

Relation bien-fondée

Aucune différence n'apparaît entre cette méthode et celle du travail. Toutefois, ce ne sont plus les constructeurs des TDAs que l'on manipule, mais les constructeurs offerts par le langage de la logique des prédicats du premier ordre.

Forme structurelle du paramètre d'induction

Nous devons construire la conjonction de littéraux pour toutes les différentes formes structurelles du paramètre d'induction. Ceci revient à dire qu'il faut déterminer quels sont les littéraux valides pour le paramètre d'induction.

Chaque élément de l'ensemble repris dans les C_i est supposé être quantifié existentiellement (partie définition d'un algorithme logique). C_1, C_2, ..., C_n doivent couvrir toutes les formes possibles du paramètre d'induction : pour n'importe quelle instanciation x du paramètre, nous avons $X=t_1$ ou $X=t_2$ ou ... ou $X=t_n$ dans n'importe quel modèle de Herbrand.

exemple :

Littéraux possibles pour la séquence, selon la relation d'ordre bien fondé choisie :

forme fixée [] : séquence vide
forme récursive [E|S] : séquence d'au moins un élément (d'un élément et d'une sous-séquence)

Construction des Formules sémantiquement équivalentes au prédicat

Chaque Fi doit être une condition nécessaire et suffisante pour que le prédicat soit vérifié (vrai) pour une instance des paramètres satisfaisant les préconditions de la spécification quand Ci appartient à un modèle de Herbrand.

exemple :

Soit la fonction *del* de suppression d'un élément d'une liste (séquence) S qui est en position N.

Le prédicat correspondant *p-del(N,S,Sr)* est vrai si la séquence Sr est la suite des éléments de la séquence S, excepté pour le Nième élément qui n'est pas repris.

Paramètre d'induction retenu : N
Relation d'ordre bien fondé : n1 < n2 ssi n1 est une valeur plus petite que n2 dans l'ensemble mathématique N.

Formules logiques obtenues :

p-del(1,[E|S1] ,Sr) ⇔ Sr=S1
p-del(N,[E|S1],[E|Sr]) ⇔ N1 = N-1 & p-del(N1,S1,Sr)

5. Brève numération des différences dans les deux méthodes

Tentons de mettre en évidence quelques différences notables entre les deux méthodes de construction par induction structurelle.

a. Différence de conception

Les deux méthodes diffèrent par l'objet de leur démarche. Dans la méthode logique, on tente de définir la réalité (ou l'existence) d'un prédicat en construisant des algorithmes logiques. Dans la méthode fonctionnelle, la définition porte sur la valeur d'un résultat que la fonction à spécifier doit calculer.

Soit **p-fonction** : le prédicat de l'algorithme logique
f-fonction : le foncteur en premier argument de l'égalité sémantique d'un axiome équationnel

$$\textbf{f-fonction}(t_1,...,t_n) == t_x$$
$$\textbf{p-fonction}(t_1,...,t_n,t_x) \Leftrightarrow \textbf{Def}$$

b. Différence de concision des formules obtenues

Une formule fonctionnelle est bien plus concise qu'une formule logique, puisqu'elle n'est exprimé que par un seul terme ! De là, la lecture y est plus difficile, puisqu'il faut refaire mentalement l'évaluation des résultats intermédiaires.

c. Différence de clarté procédurale

Dans la méthode logique, aucun ordre n'est imposé dans les termes du corps de la clause. En effet, on ne manipule que des prédicats, sans vouloir connaître quel est l'ordre d'évaluation des résultats intermédiaires. Dans la méthode fonctionnelle, l'ordre est dicté par la profondeur de la fonction dans le terme. Ainsi, il faut d'abord évaluer les fonctions de niveau N-1 pour pouvoir calculer le résultat de la fonction N.

DEUXIEME PARTIE : LA PROGRAMMATION LOGIQUE DE SPECIFICATIONS ABSTRAITES

L'importance du choix d'un langage pour la réalisation pratique des algorithmes abstraits ne fait aucun doute. Dès que la phase d'écriture des spécifications formelles a été terminée, un langage d'implémentation des axiomes équationnels a été retenu : *Prolog*.

Les axiomes abstraits définissent le sens de fonctions caractérisant un type abstrait. Nous avions la ferme intention de respecter l'idée fonctionnelle - c'est-à-dire l'optique "obtention de résultats calculés sur base d'un ensemble complets d'arguments" - des opérations spécifiées. Elles seraient idéalement programmés fonctionnellement. Or, la programmation logique, dont Prolog est le langage le plus connu, partage de nombreuses caractéristiques avec la programmation fonctionnelle.

Toutefois, nous aimerions, ultérieurement, implémenter les spécifications à l'aide d'un langage fonctionnel tel que le ML standard (réf 31 de DeGroot et Lindström).

Suite au choix et à la présentation théorique du Prolog (ainsi que de ses différences avec les langages fonctionnels), nous nous sommes intéressés à la préparation d'une méthode de transformation systématique des axiomes abstraits en clauses Prolog. Le cinquième chapitre explicite les étapes nécessaires à passer pour cette transformation.

CHAPITRE 4 : La programmation logique et le langage Prolog

A. Introduction

Un intérêt croissant est apparu ces dernières années pour ce que l'on a qualifié de langages déclaratifs. Ces langages représentent une nouvelle tendance radicale s'écartant des langages conventionnels (dits aussi impératifs). Le bien fondé mathématique des langages déclaratifs a apporté certains bénéfices tels qu'une puissance d'expression accrue, la possibilité de manipulations formelles, et la facilité de l'évaluation parallèle, ce qui rend possible l'émergence de la nouvelle génération d'ordinateurs basés sur ces langages.

Dans ces langages déclaratifs, dominent pour l'instant deux écoles principales :

- les langages fonctionnels, qui retrouvent leurs origines dans le calcul lambda et dans les systèmes d'équations récursives, et,
- les langages logiques, fondés sur l'interprétation procédurale du calcul des prédicats du premier ordre.

Bien que les deux styles présentent les mêmes bénéfices dus à leur nature déclarative, il existe quand même d'importantes et fondamentales différences stylistiques, que nous allons tracer brièvement au point suivant, avant de n'envisager que la programmation logique, et son leader, le Prolog.

B. Comparaison des langages fonctionnels et logiques

Depuis que la popularité de la programmation logique s'est affirmée par le biais de Prolog, les débats sur les aspects positifs et négatifs opposant les "pro-logiques" et les "pro-fonctionnels" ont fait surface. DeGroot et Lindström (1986) en ont fait une brillante synthèse, présentant des articles de U.S. Reddy et Darlington et al, dans la première partie de leur livre.

1. Aspects communs

Les deux formes de programmation partagent les caractéristiques suivantes :

- une nature applicative (manipulation de valeurs plutôt que de cellules assignables),
- un intérêt dans la récursion pour la programmation modulaire et,
- des facilités pour le parallélisme d'exécution.

2. Différences notables

D'après Darlington et al (in DeGroot, Lindström '86), un programme fonctionnel moderne est un ensemble d'équations définissant des fonctions. L'exécution d'un tel programme met en oeuvre la réduction d'une expression par l'utilisation des équations comme des règles de réécriture de gauche à droite, jusqu'à ce qu'elle ne contienne plus de fonctions définies, mais uniquement des constantes et des fonctions de construction.

Cette définition, bien que non généralisable à tous les langages fonctionnels (LISP par exemple), correspond idéalement au cadre de ce travail.

Tous les langages fonctionnels matures sont du plus haut ordre, cela signifie que les fonctions sont des objets de première classe et peuvent être passés comme paramètres et retournés comme valeurs. En outre, ces langages sont typés. Nous comprenons dès lors l'intérêt de ces langages pour l'implémentation des spécifications en type abstrait.

a. Typage

Prolog et associés (respectant le standard) sont actuellement tous non typés, alors que les langages fonctionnels sont fortement typés et permettent le polymorphisme.

b. L'ordre supérieur

Les langages logiques sont des langages du premier ordre, alors que les langages fonctionnels ne le sont pas. Bien sûr, la possibilité d'écriture de fonctions de l'ordre supérieur augmente la puissance d'expression d'un langage, permettant à plusieurs algorithmes liés d'être réalisés comme des instances spécifiques d'une unique fonction générique (et diminuant la taille du programme, ce qui en facilite sa lecture).

c. Notation fonctionnelle ou relationnelle

La sortie (résultat) de chaque fonction d'un programme fonctionnel est définie implicitement par la construction dans la partie droite de l'équation. Une définition équivalente dans un langage logique aurait nécessité que chaque composant de sortie soit nommé dans la tête de clause.

De plus, des variables supplémentaires doivent être introduites dans le corps de la clause pour représenter ce que le programme fonctionnel fait par simple composition de fonctions. Sachant que l'invention de noms pour les variables est une tâche ardue en programmation, on comprend que la productivité s'en ressente.

exemple :

nous utilisons, dans cet exemple, un constructeur de base d'une liste qui est []. L'effet de celui-ci est de pouvoir accéder à la liste soit par son élément de tête, soit par la sous-liste des éléments suivants (append).

notation fonctionnelle (dans un langage conceptuel)

```
reverse :   SEQ(ELEM) --> SEQ(ELEM)

        reverse([]) <= []
        reverse([H|T]) <= conc(reverse(T),[X])

conc :   SEQ(ELEM), ELEM --> SEQ(ELEM)

        conc([],X) <= [X]
        conc([H|T],X) <= [H|conc(T,X)]
```

notation logique (en langage Prolog)

```
        reverse([],[]).
        reverse([H|T],Lr)  :-  reverse(T,Lr1),  conc(Lr1,[X],Lr).

              (*     introduction de la variable Lr1   *)

        conc([],X,[X]).
        conc([H|T],X,[H|Lr])  :- conc(T,X,Lr).
```

Remarque :

Bien que l'écriture de reverse ou de conc est la même dans les deux notations, ces opérations sont conceptuellement différentes. Les préfixes f- et p- permettent de faire la distinction entre la notation fonctionnelle qui spécifie des foncteurs (f-reverse et f-conc), et la notation logique qui spécifie des prédicats (p-reverse et p-conc).

f-conc est une fonction qui retourne un résulta correspondant à l'application de la fonction de concaténation à deux arguments séquence.

p-conc est un prédicat qui est vrai si le troisième argumen séquence est bien le résultat de la concaténation des deux premiers arguments de type séquence.

d. Emploi mutli-mode des relations

La notation et la résolution relationnelle offre aux langages logiques une caractéristique très élégante et très puissante, que n'ont pas les langages fonctionnels. Cela est dû au fait que le programme logique ne fait pas de conditions sur le nombre et le type de variables qui doivent être instanciées dans la question. Une relation définie une fois pour toute peut servir dans différents modes d'utilisation. Ainsi, un programme logique peut déterminer, à partir d'un résultat de fonction, l'ensemble des valeurs possibles pour les autres arguments de cette fonction (par exemple, pour obtenir un ensemble de jeux de tests).

De là, la programmation logique est plus expressive, représentant en une seule fonction plusieurs autres.

e. Sorties non instanciées

La programmation logique ne nécessite pas que l'entièreté des variables soient instanciées lors de l'utilisation de la procédure. Le mécanisme d'exécution du programme tentera d'instancier ces variables. Si il n'y arrive pas, c'est soit qu'il y a échec du programme, soit que les variables sont inchangées, représentant un ensemble générique de valeurs.

Quant aux langages fonctionnels, ils sont restreints à l'instanciation obligatoire aux constantes et constructeurs. Cela limite leur puissance.

f. Déterminisme

Les langages fonctionnels sont déterministes : les seules formes syntaxiques permises sont celles qui exprime une fonction selon la définition mathématique, c'est-à-dire ayant un et un seul résultat calculé sur base d'un ensemble de valeurs en entrées. L'exécution d'un programme fonctionnel qui se termine apportera toujours une solution unique à la fonction principale.

En contraste, chaque question en programmation logique apportera une ou plusieurs solutions, ou même parfois aucune solution. Ceci est dû à l'incomplétude des stratégies de recherche de solutions implémentées. En outre, même si une stratégie de recherche complète est employée, différents choix dans l'espace de recherche peuvent avoir différentes conséquences parfois très vastes. Par exemple, même si une solution existe, il est possible que l'exécution du programme ne la trouve pas suite au choix d'une recherche en profondeur d'abord. Ce problème est détaillé plus loin.

g. Origine des différences

Les différences évoquées ci-dessus proviennent, à n'en pas douter, du mécanisme d'appel qu'utilisent les deux types de langages.

Les langages fonctionnels emploient le *pattern-matching* ou, en français, appariement des formats, tandis que les langages logiques utilisent l'*unification* .

Ne détaillons pas plus les caractéristiques de chaque mécanisme, ces termes seront expliqués plus loin.

La conclusion que tirent Darlington et al. (in DeGroot et Lindström, 1986) est que *"chacun de ces styles de langage a quelque chose à offrir à l'autre, et qu'un chemin vers un langage éprouvé peut être trouvé par les deux camps, en important des caractéristiques de l'autre."*

Nous passons maintenant à la programmation logique proprement dite, qui nous servira à implémenter les axiomes équationnels des fonctions abstraites. Ceci signifie que l'utilisation du Prolog se fera dans une otpique fonctionnelle proche de la spécification des types de données abstraits.

C. Historique de la programmation logique

La programmation logique est apparue relativement récemment dans la vaste étendue des branches informatiques. En effet, ce n'est qu'en 1972, avec Robert Kowalski pour la partie théorique, Maarten Van Emden pour la démonstration expérimentale, tout deux d'Edinbourgh, et Alain Colmerauer pour l'implémentation, à Marseille, que les notions de cette façon innovatrice de programmer ont été exprimées. Elles se basaient sur les travaux de Herbrand (1930), Prawitz, Gilmore, Davis et Putnam (1960) dans l'application à l'intelligence artificielle des preuves de théorèmes, ainsi que sur le fameux principe de résolution de Robinson (1965).

Kowalski décrit les algorithmes logiques par deux aspects : d'une part, la logique, c'est-à-dire ce qu'est le problème à résoudre, et le contrôle, comment il faut le résoudre. Ainsi, le contrôle est l'interprétation procédurale de la logique. Ces aspects sont totalement déclaratifs.

Dès lors, la logique pouvait être utilisée comme langage de programmation. Il restait à mettre en oeuvre, sur machine, ce type de langage. Roussel fut le premier, toujours en 1972, en écrivant en Algol-W le premier interpréteur Prolog, qui a fait de nombreux petits frères depuis, tous plus performants que les précédents. Mais Prolog doit largement sa popularité actuelle à l'implémentation efficace de David Warren, vers 1975, à Edinbourgh.

Toutefois, ces versions semblent s'écarter petit à petit de l'aspect purement déclaratif du langage, par l'adjonction de procédures d'entrée/sortie, et de procédures extra-logiques.

D. Qu'est-ce que la programmation logique ?

Voici la définition qu'en donne Sterling et Shapiro, dans leur livre intitulé "The art of Prolog" (1986) : *"Un programme logique est un ensemble d'axiomes, ou règles, définissant des relations entre objets. Une exécution d'un tel programme est une déduction des conséquences du programme. Un programme détermine, par là, un ensemble de conséquences, qui forment sa signification. Tout l'art de la programmation logique est de construire des programmes élégants et concis qui ont la signification souhaitée"* .

E. Qu'est-ce que le Prolog ?

Prolog est un langage de programmation de type logique, centré autour d'un petit ensemble de mécanismes de base, comportant le pattern-matching, l'unification des clauses, la structuration des données en arbre, et le backtracking automatique. Ce petit ensemble constitue un cadre de programmation extrêmement puissant et flexible. Prolog est spécialement bien adapté pour les problèmes concernant les objets - en particulier, les objets structurés - et leurs relations.

Grâce à l'écriture d'un ensemble de règles, et à l'introduction de questions, l'interpréteur Prolog tente de vérifier si les questions sont solubles dans le contexte défini. Ceci en fait un langage efficace pour l'intelligence artificielle et pour la programmation non-numérique en général.

Notre choix s'est porté sur le langage Prolog parce qu'il offre les avantages suivants :

- syntaxe proche de celle des FOPL, et d'ailleurs fondée sur cette dernière,
- emploi intensif de la récursion,
- programmation en module facile (pas de notion d'ordre, à priori)

F. La déclaration des objets et de leurs relations en Prolog

Le langage Prolog est, dans sa forme standard, très proche du langage des prédicats du premier ordre, duquel il s'inspire. Il n'y a donc que peu de différences au niveau de la base formelle théorique des termes employés. Nous présentons ci-dessous de façon informelle les termes décrits préalablement formellement dans le deuxième chapitre.

1. Constructions de base

Les constructions de base de la programmation logique, termes et instructions, sont dérivées de la logique mathématique. Les termes Prolog (variables, constantes, faits, questions et règles) sont syntaxiquement identiques aux termes définis dans le chapitre traitant des prédicats du premier ordre (variables, constantes, foncteurs et prédicats). Il y a trois sortes d'instructions composant un programme logique : les faits , les règles et les questions.

a.Constantes

Les constantes Prolog sont toute entité représentant un objet de la réalité de façon unique et invariable. les identificateurs suivants sont des constantes : poisson, pi, un, 1. Ils doivent obligatoirement débuter par une minuscule ou un chiffre.

b.Variables

Une variable logique exprime une individualité non spécifiée et est utilisée de la même façon qu'une individualité spécifiée. C'est un moyen permettant de généraliser un ensemble de termes dont le destin est identique. Ces variables ont un comportement différent des variables des programmes conventionnels, car ils représentent une entité simple non spécifiée plutôt qu'une adresse de stockage en mémoire. Le premier caractère de l'identificateur d'une variable Prolog doit être une majuscule. Nom_du_pere et Nom_du_fils sont deux variables

c. Termes

Un terme Prolog est défini récursivement : c'est une constante ou une variable, ou une structure qui est une composition de termes au sein d'une fonction. Dans ce dernier cas, le terme est caractérisé par un nom (atome), encore appelé foncteur, et une arité (nombre d'arguments). Cette définition du terme est identique à celle donnée formellement au deuxième chapitre.

Quand dans un terme n'intervient aucune variable, on dit de celui-ci qu'il est **clos** ("**ground**" en anglais), puisqu'il représente une valeur de la réalité unique et connue.

Le principe de substitution

Déf. Une substitution θ est un ensemble fini (éventuellement vide) de paires de la forme :

$$X_i = t_i, \quad \text{où} \quad X_i \in \text{VARIABLE, et}$$
$$t_i \in \text{TERME}$$

avec $\forall i,j;\ X_i <> X_j$ et $\forall i,j;\ X_i$ n'apparaît pas dans t_j

Les substitutions peuvent être appliquées aux termes. Le résultat de l'application d'une substitution θ sur un terme t ($t\theta$) est le terme obtenu par remplacement de chaque occurence de X_i par t_i dans t, pour chaque paire $X_i=t_i$ de θ.

La notion d'instance

Déf. t_1 est une instance de t_2 si il y a une substitution θ telle que $t_1 = t_2\theta$, où $t_1, t_2 \in$ TERME

Déf. si t_3 est une instance de t_1
et t_3 est une instance de t_2
Alors t_3 est une instance commune de t_1 et t_2,
où $t_1, t_2, t_3 \in$ TERME
Donc il existe des substitutions θ_1 et θ_2 telles que $t_3 = t_1\theta_1 = t_2\theta_2$ (égalité syntaxique).

Les termes Prolog sont de trois types : fait, question ou règle, que nous allons détailler un à un.

d. Faits

Le fait est la forme la plus simple d'une instruction. c'est un moyen d'établir une relation entre objets. La relation est un prédicat, et les noms caractérisant les objets du prédicat sont des atomes (au sens FOPL).

Un ensemble fini de faits Prolog constitue un programme Prolog. C'est la forme la plus simple du programme logique. De façon intuitive, on appelle ce programme "une description de situation". Il est valable de dire que ceci forme une base de données relationnelle.

Syntaxe :

$p(t_1,...,t_n)$. avec $t_1,...,t_n \in$ TERME

Sémantique :

un fait p(t1,...,tn). établit la vérité de la relation p entre ses arguments t1,...,tn.

$\forall X_1,...,X_k$, où $X_i \in t_j$, p(t1,...,tn) est vrai.

e. Questions

Les questions, seconde forme d'instruction d'un programme logique, sont un moyen de "dénicher" les informations du programme.

Question simple

Une question est un fait dont on désire vérifier la cohérence dans un ensemble de faits déjà établis. C'est une vérification de l'existence de relations entre objets. Syntaxiquement, la question Prolog est identique au fait Prolog, bien que le point final est remplacé par un point d'interrogation.

En termes plus rigoureux, répondre à une question, c'est vérifier que la question est une conséquence logique du programme.

Les conséquences logiques sont obtenues par des règles de déduction. Ce n'est donc pas assurer une quelconque vérité sur la question autre que celle citée ci-dessus.

Syntaxe :

$p(t_1,...,t_n)$? **avec $t_1,...,t_n \in$ TERME**

Sémantique :

une question p(t1,...,tn) ? demande que l'on vérifie si la relation p est établie entre les arguments t1,...,tn.

$\exists ?\ X_1,...,X_k$, où $X_i \in t_j$,
tel que programme $\models$ p(t1,...,tn)

Une question contenant une variable demande si il y a une valeur pour la variable qui fait de la question une conséquence logique du programme. Le principe de substitution est mis en oeuvre pour obtenir ces valeurs à partir du programme.

Conjonction de questions

Une question peut être formée par la conjonction de questions plus simples. Dans ce cas, chaque question doit être une conséquence logique du programme pour que la question principale soit vérifiée.

Syntaxe :

$Q_1,...,Q_n$? avec $Q_1,...,Q_n \in$ QUESTION

Sémantique :

$\exists ?\ X_1,...,X_k$, où $X_i \in t_j$, et $t_j \in Q_k$,
tel que $P \models Q_1,...,Q_n$, avec $P \in$ PROGRAMME.
ou, autrement dit,
$P \models Q_1$
et ...
et $P \models Q_n$.

Dans le cas d'une conjonction de questions, qu'en-est-il des variables partagées par certaines questions ? Les variables partagées sont un moyen de contraindre une question en restreignant le domaine de la variable à un sous-ensemble de valeurs encore possibles. Une variable de ce genre doit être instanciée aux mêmes valeurs dans chaque question de la conjonction pour que cette dernière puisse être une conséquence logique du programme.

Opérationnellement, résoudre $Q_1,...,Q_n$? en utilisant un programme P, c'est trouver une substitution θ telle que $Q_1\theta$ et ... et $Q_n\theta$ soient des instances closes de faits de P.

Les règles de déduction

Les règles de déduction sont le moyen d'analyser la validité d'une question selon un programme logique.

REGLE N°1 : IDENTITE

$p \models p$ (De p, déduire p)

Une question est une conséquence logique d'un fait si l'ensemble des faits (ou programme) possède un fait qui est identique à la question.

La réponse à la vérification de validité de la question selon l'ensemble des faits est 'Oui' si il existe un fait identique à la question dans le programme, et 'Non' sinon.

REGLE N°2 : GENERALISATION

$p\theta\ \forall\theta\ |= \forall X_i\ p(X)$, où θ est une substitution portant sur les variables X_i de la question p.

Une question existentielle p est une conséquence logique d'une instance $p\theta$, pour toute substitution θ .

La réponse à la vérification de validité d'une question existentielle non close selon l'ensemble des faits est l'instance de la question dans cet ensemble, si il existe un fait qui est une instance de la question, à un jeu de substitution près, 'Non' sinon.

REGLE N°3 : INSTANCIATION

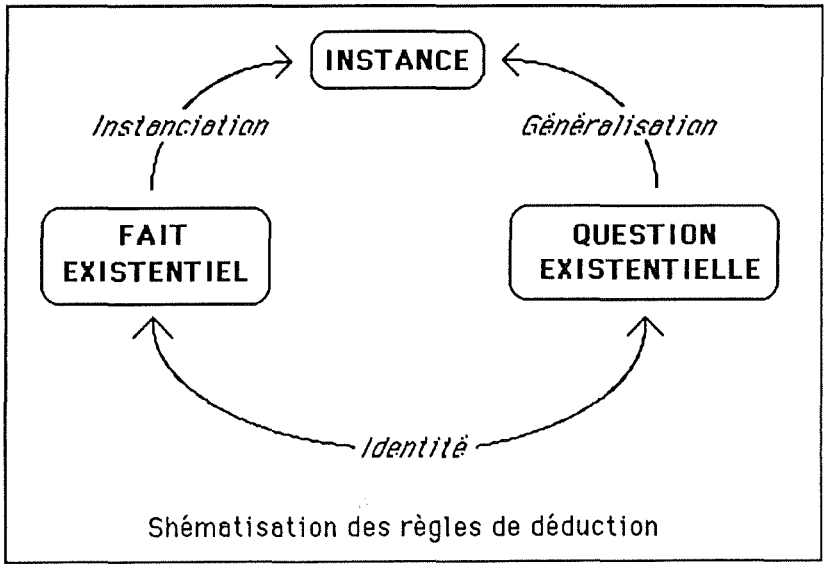
$\forall X_i\ p(X) |= p\theta\ \forall\theta$, où θ est une substitution portant sur les variables X_i du fait p.

D'un fait p quantifié universellement, on peut déduire chaque instance $p\theta$, pour chaque substitution θ.

La réponse à la vérification de validité d'une question close selon un ensemble de faits quantifiés universellement est le fait pour lequel la question est une instance dans cet ensemble, si il existe, 'Non' sinon.

CONCLUSION

La réponse à la vérification de validité d'une question selon un ensemble de faits est l'instance commune à la question et à un des faits de l'ensemble, si elle existe, 'Non' sinon.



Shématisation des règles de déduction

f. Règles

La troisième, et probablement la plus importante, des formes de termes Prolog est la règle. Celle-ci permet de définir de nouvelles relations à l'aide d'autres relations préexistantes. Sa syntaxe correspond à celle de la plus générale des clauses logiques.

Syntaxe :

$t <- t_1,...,t_n.$ où $n >= 0$, et $t,t_1,...,t_n \in$ ATOME

On dira que t est la tête de la règle, et t1,...,tn en est le corps. Nous voyons maintenant clairement qu'une question ou un fait sont des cas particuliers de la règle, tous trois étant des clauses de Horn, ou clauses tout simplement. Une clause ayant un corps d'un seul terme est une clause itérative.

Comme pour les faits, les variables apparaissant dans les règles sont quantifiées universellement, et leur portée est étendue à toute la règle.

Sémantique :

La sémantique d'une règle quelconque est double, selon le point de vue de l'utilisateur ou du programmeur de la règle. Nous prendrons la règle générale suivante pour expliciter ces sémantiques :

$$A(t) <- B_1(t_1),..., B_n(t_n).$$

Expression déclarative d'une règle

Une règle est un axiome logique, dans laquelle le symbole '<-' correspond à l'implication logique.

L'interprétation déclarative de la règle est : "Pour toutes les variables de la règle, la relation A est vraie si les relations B_1 et ... et B_n sont vraies.

Bien que toutes les variables d'une clause soient quantifiées universellement, nous pouvons faire référence à certaines variables présentent uniquement dans le corps de la clause, comme si elles étaient quantifiées existentiellement dans cette partie.

L'interprétation déclarative de la règle devient, dans ce cas, : "Pour toutes les variables de la tête de règle, la relation A est vraie si il existe des valeurs pour les variables du corps de règle telles que les relations B_1 et ... et B_n sont vraies.

Expression procédurale d'une règle

Une règle est un moyen permettant d'exprimer de nouvelles questions, ou des questions complexes, en terme d'autres questions. Ceci respecte le principe de réduction de problèmes en sous-problèmes plus simples, permettant d'obtenir un résultat.

L'interprétation procédurale de la règle est : "pour vérifier la question 'Y-a-t'il une relation A entre ses arguments t ?', il faut vérifier qu'il existe une relation B_1 entre les arguments t_1, et ..., et une relation B_n entre les arguments t_n".

Une nouvelle règle de déduction : le Modus Ponens

La loi du Modus Ponens universel dit que :

De la règle $R = A <- B_1,..., B_n.$
et des faits $B'_1.$
...
$B'_n.$

On peut déduire la tête de règle A' si $A' <- B'_1,..., B'_n.$ est une instance de R.

Cette loi inclut comme cas particuliers la déduction par identité et par instanciation.

2. Le programme déclaratif Prolog

a. Syntaxe d'un programme

Nous terminerons la description de la notion déclarative de la programmation en Prolog, qui ne tient d'auncun aspect de fonctionnalité ou d'optimisation, par les définitions suivantes.

Déf. Un *programme logique Prolog* est un ensemble fini de règles.
Une *procédure Prolog* est le sous-ensemble des règles du programme dont le prédicat est le même.

En ce qui concerne la notion de procédure, Sterling et Shapiro (1986) ont montré qu'elle est identique à celle des styles de programmation classique, si on considère l'interprétation opérationnelle de ces règles par l'interpréteur Prolog.

b. La sémantique d'un programme logique

La sémantique d'un programme Prolog P, M(P), est l'ensemble des termes unitaires et clos que l'on peut déduire de P. Lorsqu'un programme n'est composé que de faits clos, alors la signification du programme est implicite, c'est le programme lui-même.

c. La notion d'exactitude d'un programme

Afin de clarifier un peu plus la sémantique d'un programme logique, ajoutons l'explication de l'exactitude et de la complétude, de façon informelle (la littérature de la programmation logique détaille suffisamment ces notions mathématiquement).

Déf. Un programme est *cohérent* par rapport à une sémantique attendue M si la sémantique de P, M(P), est un sous-ensemble de M.

Un programme correct ne parle pas de choses non voulues.

Déf. Un programme est *complet* par rapport à une sémantique attendue M si M est un sous-ensemble de la sémantique de P, M(P).

Un programme complet parle de tout ce que l'on attend.

Déf. Un programme est *correct* par rapport à une sémantique attendue M si M est identique à la sémantique de P, M(P).

Un programme correct parle uniquement de ce que l'on attend.

G. Le mécanisme d'exécution d'un programme Prolog

Afin de présenter l'interpréteur Prolog relativement simplement, nous verrons d'abord comment fonctionne un interpréteur de programme abstrait très simple. Ensuite, nous détaillerons l'interpréteur Prolog et les extensions qu'il demande afin d'assurer certains critères de solvabilité et d'optimisation.

1. Un interpréteur abstrait simple

L'interpréteur abstrait présenté à la page ci-contre est très simple, de par les restrictions qu'on lui impose :

- Seul le Modus Ponens universel appliqué à des termes clos est permis,
- A partir d'un programme P et d'une question Q, le résultat de l'interprétation sera "OUI" si Q peut être prouvé par (est une conséquence logique de) P, et "NON" autrement.
- En cas d'échec de terminaison, si Q n'est pas prouvable par P, il n'y aura pas de réponse du tout.

Déf. Le *résolvant* est l'objectif courant à n'importe quelle étape du calcul.

Déf. La *trace* de l'interpréteur est la séquence des résolvants produits lors du calcul, et selon les choix qui ont été faits.

Déf. Une *réduction fondée* d'un terme G par un programme P est le remplacement de G par le corps d'une instance fondée d'une règle de P, dont la tête est identique au terme choisi.

La réduction est l'étape de base du calcul en programmation logique. Elle correspond à une application du Modus Ponens universel.

L'algorithme de l'interpréteur abstrait simple est présenté sur la page de gauche, sous la forme d'un graphe de Nassi et Schneidermann.

L'interpréteur propose deux choix dans son algorithme, sans les spécifier :

a) Quel terme du résolvant est à réduire en premier lieu ?
b) Quelle clause, et quelle instance fondée sont à prendre en premier ?

En ce qui concerne le choix du terme du résolvant, l'ordre est arbitraire, car dans les résolvants, tous les termes devront être réduits, et l'ordre de réduction n'a aucune importance dans le cas de preuve.

Par contre, le choix de la clause est critique. Deux possibilités existent : le déterminisme et le non-déterminisme. Le second cas est un choix non spécifié, à partir d'un certain nombre d'alternatives, qui est supposé éclaircir le problème. Quant au déterminisme, il s'agit d'un choix orienté selon des connaissances sur la solution au problème. Dans les deux cas, aucune machine ne peut implémenter directement ces concepts, mais le non-déterminisme peut être approximé, comme nous le verrons pour le Prolog.

Le fonctionnement de l'interpréteur

L'exécution de l'algorithme de l'interpréteur permet de créer un arbre de recherche (également dit arbre de résolution) et/ou dans lequel les unifications (dont l'explication et le mécanisme sont fournis plus loin) sont présentes de haut en bas, et les termes du résolvant de gauche à droite. Les feuilles de l'arbre sont soit les faits (et dans ce cas, il y a réussite du chemin qui y mène), soit les termes du résolvant pour lesquels on cherche une clause qui s'unifie (Si on n'en trouve pas, il y a échec du chemin qui y mène).

Afin de déterminer les solutions, l'interpréteur "devine" les chemins qui réussissent (jusqu'aux faits). Deux méthodes de parcours de l'arbre de recherche existent :

PARCOURS EN LARGEUR D'ABORD

Tous les choix possibles sont explorés en parallèle. Cela garantit que l'interpréteur trouvera une preuve finie, si elle existe. C'est une *procédure de preuve complète.*

PARCOURS EN PROFONDEUR D'ABORD

Ce parcours ne garantit pas que l'interpréteur trouvera une preuve finie, même si elle existe, car l'arbre peut avoir des branches infinies, correspondant à l'exécution "potentiellement" infinie de l'interpréteur non-déterministe. Alors, l'interpréteur bouclera dans cette branche, alors que la solution peut être dans une branche suivante. C'est une *procédure de preuve incomplète.*

Condition d'arrêt

Un calcul de G par P se termine si G_n est vrai ou échoue, pour certains n>=0. Ce calcul est fini et de longueur n. Un calcul réussit quand il se termine avec la valeur vraie.

2. Généralisation de l'interpréteur abstrait aux questions non fondées.

Voici l'adaptation de l'interpréteur abstrait de programmes logiques. La restriction aux instances fondées de clauses est supprimée. De plus, l'algorithme d'unification est appliqué aux termes du résolvant choisis et à la tête de la clause retenue afin de trouver une substitution correcte à effectuer sur le nouveau résolvant.

a. Fonder la question

L'interpréteur est étendu afin de répondre aux questions existentielles non fondées, grâce à une phase d'initialisation additionnelle qui permet de travailler avec une instance fondée de la question (voir page de garde). Cette phase est identique à la phase de choix des instances fondées des règles (ou clauses) de l'interpéteur. La difficulté, en général, à deviner des instances fondées correctes découle du fait qu'il faudrait connaître à l'avance le résultat de l'exécution.

b. Le mécanisme d'unification

Le mécanisme d'unification est à la base de la déduction automatique. Il détermine quel est l'unifieur le plus général de deux termes, si il existe, et rapporte son échec sinon. Définissons ce que l'on entend par unifieur le plus général.

Déf. Un *unifieur* de deux termes est une substitution rendant les deux termes identiques. Si les deux termes ont un unifieur, ils s'unifient. De là, l'unifieur détermine une instance commune (et vice-versa).

Déf. L'*unifieur le plus général* de deux termes est un unifieur tel que l'instance commune associée est la plus générale.

Une instance de la question pour laquelle une preuve est trouvée est appelée la *solution* à la question.

c. Remarque sur la portée des variables

Les variables sont locales à une et une seule clause. Ainsi, les variables de différentes clauses qui ont le même nom sont, en réalité, différentes, même au sein d'un procédure.

3. Le Prolog pur

Prolog illustre deux réalités complémentaires. La première est que Prolog est un langage de programmation permettant de déclarer un programme logique, dans lequel il définit un ordre pour les clauses dans une procédure, et pour les termes dans le corps d'une clause. La seconde est que Prolog est une réalisation concrète, spécialisée, de l'interpréteur abstrait qui tient compte de ces informations ordonnées.

a. Le modèle Prolog d'exécution

Deux décisions majeures doivent être prises pour concrétiser l'interpréteur abstrait :

a. Le choix du terme du résolvant, et
b. Le choix non déterministe de la clause.

Déf. Le mécanisme d'exécution Prolog est obtenu à partir de l'interpréteur abstrait :

- en choisissant le terme le plus à gauche dans le résolvant,
- en remplacant le choix non déterministe d'une clause par une recherche séquentielle d'une clause unifiable (de haut en bas), avec retour en arrière.

Un calcul d'une question G selon un programme P est la génération de toutes les solutions de G selon P. En terme de concepts de programmation logique, il s'agit d'une traversée en profondeur d'abord dans l'arbre de résolution de G, obtenu en choisissant le terme le plus à gauche.

b. Le retour en arrière ('Backtracking')

Le retour en arrière, dans une branche de l'arbre de résolution, suite à un échec, ou à la découverte d'une solution de G, peut être caractérisé de :

Superficiel : quand l'unification d'un terme et d'une clause échoue, et qu'il faut essayer une clause alternative, ou de

Profond : quand l'unification d'un terme avec la dernière clause d'une procédure échoue, et qu'il faut retourner au dernier terme précédent la réduction.

c. L'ordre Prolog des règles

Déf. L'ordre des règles détermine l'ordre dans lequel les solutions seront trouvées

Cette définition signifie que changer cet ordre revient à permuter les branches dans l'arbre de recherche. Ceci n'a aucune influence sur les feuilles de l'arbre, c'est-à-dire les faits. Leur ordre n'a donc aucune importance.

Conséquences :

a) L'ordre des solutions des questions résolues par un programme récursif est déterminé par l'ordre des règles.

b) Quand un arbre de recherche pour un terme donné a une branche infinie, l'ordre des règles peut déterminer si toutes les solutions seront données, ou si le calcul ne se terminera pas. Puisque Prolog choisit une traversée en profondeur d'abord, l'interprétation pourrait cycler dans une branche infinie. Ainsi, changer l'ordre des branches avance ou retarde le moment de bouclage.

d. L'ordre Prolog des termes d'une clause

Cet ordre est plus significatif ! C'est le moyen principal de spécifier le flux séquentiel de contrôle dans les programmes Prolog.

Déf. L'ordre des termes composant le corps d'une clause détermine l'arbre de recherche.

Suite à cette remarque, il est clair que l'ordre des termes détermine la terminaison ou non du calcul, et ceci est d'autant plus sensible que la récursivité joue également : en effet, les règles récursives dont le terme récursif est à gauche, posent plus de problèmes. Et comme en Prolog, il vaut mieux que le terme échoue le plus rapidement possible, taillant ainsi l'arbre de résolution plus tôt, l'heuristique à promouvoir est de placer le terme récursif le plus à droite.

4. Conclusions

Tenant compte des critères de choix imposés par Prolog, la programmation déclarative s'en ressent. Idéalement, il fallait écrire des axiomes qui définissent des relations tout en ignorant la façon dont ils

sont utilisés par le mécanisme d'exécution. La programmation logique effective demande, malheureusement, de connaître et d'utiliser les choix d'implémentation.

En outre, les conséquences sur les performances de l'interpréteur Prolog sont énormes : bien souvent, pour des cas habituels, un programme logique correct échouera dans sa tentative de résolution à cause de la non-détermination.

H. Utilisation fonctionnelle du Prolog

Dans l'intention volontaire de conserver l'aspect fonctionnel de la spécification à l'aide des TDAs, nous avons focalisé notre étape de transformation des spécifications en procédures logiques sur la programmation de procédures logiques fonctionnelles.

Formellement, ceci revient à assurer que tous les arguments de la clause question soient instanciés, excepté pour celui qui joue le rôle de résultat. La notation ci-dessous illustre cette idée :

$q(t_1,...,t_{n-1},X_n)$?, avec $X \in$ VARIABLE,
$t \in$ TERME | t est fondé

ou encore,

in : *gr* *gr* *var*
$q(t_1,...,t_{n-1},X_n)$?
out: *gr* *gr* *gr*

avec *gr* exprimant que le terme est fondé
var non fondé

L'écriture des symboles *gr* et *var* au-dessus de la clause détermine l'état des termes avant utilisation de la clause (entrée ou input) et en-dessous de la clause détermine l'état des termes après utilisation de la clause (sortie ou output).

L'article "Equality for Prolog" de W.A. KORNFELD dans DeGROOT et LINDSTROM (1986) présente un aspect intéressant de l'utilisation du Prolog, avec tendance à la fonctionnalité. Signalons-en les grandes lignes.

L'extension du langage Prolog, appelée "Prolog avec égalité" permet l'inclusion d'assertion (de règles) concernant l'égalité. Quand une tentative d'unification de deux termes qui ne s'unifient pas syntaxiquement est faite, un théorème d'égalité peut être utilisé pour essayer de prouver que

les deux termes sont égaux. S'il est possible de prouver que les deux termes sont égaux, l'unification réussit, avec l'instanciation des variables introduites par la preuve d'égalité.

Cet ensemble de théorèmes d'égalité est une procédure unique avec comme prédicat "eval", pour évaluateur. En effet, dans un but logique, un autre nom eut suffit, tel "egal" par exemple. Mais l'optique fonctionnelle limitant nos investigations, tenter de vérifier si deux termes sont égaux logiquement revient à évaluer chaque terme à sa plus simple expression, et à comparer les résultats de chaque évaluation.

En outre, par cette procédure, la notation logique en clause de Prolog sera transformée en notation fonctionnelle, et assurera que le résultat de l'application du prédicat est le plus simple possible (uniquement exprimé en termes de constructeurs primordiaux).

exemple :

notation préfixée :

eval(plus(X,Y),Z) :- eval(X,X_1), eval(Y,Y_1), plus(X_1,Y_1,Z).

notation infixée :

(plus(X,Y) give Z) :- (X give X_1), (Y give Y_1), plus(X_1,Y_1,Z).

Cet article montre que la puissance de Prolog est accrue par ce mécanisme de façon significative. L'abstraction sophistiquée des données devient possible.

Maintenant, le lecteur possède tous les outils théoriques lui permettant d'accéder au chapitre présentant une méthode de transformation systématique des spécifications axiomatiques en procédures Prolog.

CHAPITRE 5 : Méthodes de transformation de spécifications algébriques en procédures Prolog

A. Principe général

La spécification algébrique d'une opération (fonction) se présente sous la forme d'un ensemble d'axiomes écrits dans le langage des prédicats du premier ordre. Quant à la procédure Prolog correspondante, ce sera un ensemble d'implications logiques. Afin de passer aisément de l'une à l'autre, il est nécessaire de transformer les axiomes selon une syntaxe proche du Prolog : la spécification sous forme conditionnelle en logique des prédicats du premier ordre (First Order Predicat Logic, FOPL), restreint à l'égalité sémantique ('==') comme seul prédicat, est le candidat retenu.

Spécification algébrique équationnelle : ensemble d'axiomes
(ADT, Algebraïc Data Types)

⇓

Spécification logique conditionnelle : ensemble d'implications
(FOPL, First Order Predicat Logic)

⇓

Procédure exécutable relationnelle : ensemble de clauses
(Prolog, Programming in Logic)

Nous allons présenter notre méthode de façon générique, afin de démontrer mathématiquement la validité de chaque étape, avant de détailler quelques transformations classiques et typées.

B. Rappels et Etape liminaire

Les opérations définies pour un type abstrait ont une spécification syntaxique et une autre sémantique. Ce sont toujours des fonctions, ayant dès lors un codomaine défini.

1. Spécification syntaxique (rappel)

Les spécifications syntaxiques TDAs peuvent être écrites de plusieurs manières . En voici deux (Nous choisirons la dernière dans ce travail, par souci de clarté) :

Syntaxe n°1 :

length : SEQ → NATURAL $\begin{cases} \text{length(S)} \in \text{NATURAL} \\ \text{S} \in \text{SEQ} \end{cases}$

ith : NATURAL, SEQ → ELEM $\begin{cases} \text{ith(N,S)} \in \text{ELEM} \\ \text{N} \in \text{NATURAL} \\ \text{S} \in \text{SEQ} \end{cases}$

Syntaxe n°2 : (en français ou en anglais)

fonction length(S) **donne** NATUREL
avec S : SEQ.
Cette fonction donne un naturel **qui est** le nombre d'éléments que contient la liste S.

fonction ith(N,S) **donne** ELEM
avec N : NATURAL,
S : SEQ
Cette fonction donne un élément **qui est** à la position N dan la liste S.

2. Spécifications sémantiques

Une opération f (fonction) est définie à l'aide d'un ensemble d'axiomes dont chacun a la forme générale suivante :

$$\overline{f(cons)} == t$$

où cons ∈ CONSTRUCT,

avec cons(u) et u ∈ (VAR ∪ CONSTRUCT), sous-ensemble de TERME. C'est donc un terme dont le symbole de tête est un constructeur de base, et dont les arguments ne peuvent être que des variables ou des constructeurs de base.

exemple :

conc(empty(),S) == S
conc(app(E,S),S_2) == app(E,conc(S,S_2))

Les axiomes prennent la forme générale suivante :

$$\overline{f(\overline{cons,X})} == t \text{ où } X \in VAR.$$

C. Etape n°1: mise sous forme clausale (FOPL)

Cette étape a pour objet de transformer les axiomes équationnels TDAs en clauses FOPL, par extraction conditionnelle des arguments. Cela nous rapprochera des clauses Prolog, et permettra d'introduire les spécifications logiques (appelées algorithmes logiques par DEVILLE).

En premier lieu, nous allons changer la représentation infixée en préfixée, afin de mettre en évidence la relation '==' d'équivalence sémantique.

$$\overline{f(\overline{cons,X})} == t$$

$$==(\overline{f(\overline{cons,X})},t)$$

Le prédicat ainsi obtenu est toujours vrai. Afin d'obtenir une forme clausale de la forme générale suivante :

$$\text{si } p_1(\overline{t_1}) \text{ alors } p_2(\overline{t_2})$$

ou

$$p_2(\overline{t_2}) \Leftarrow p_1(\overline{t_1})$$

où $p_1, p_2 \in$ PRED, $t_1, t_2 \in$ TERME, avec PRED : ensemble des prédicats logiques (fonctions dont le résultat a la valeur 'vrai').

Dans la forme générale, p_2 correspond à la relation '==', alors que p_1 est le prédicat sans argument (0-aire) *vrai()*.

$$==(\overline{f(\overline{cons,X})},t) \Leftarrow vrai()$$

ou, encore, $==(\overline{f(\overline{cons,X})},t) \Leftarrow$

1. Extension à la spécification axiomatique : les préconditions.

Il peut arriver que, pour qu'un axiome soit vrai, certaines conditions préalables (préconditions, ou conditions d'application) soient remplies.

Selon l'option prise par le spécifieur, cette précondition devra être vérifiée dans tous les cas d'utilisation de l'axiome en question, et donc devra être intrinsèque à la spécification sémantique de l'axiome. Ou alors, le spécifieur peut laisser libre choix à l'utilisateur de l'axiome, mais en précisant qu'il décline toute responsabilité en cas de non-respect de la précondition.

Dans le premier cas, nous allons permettre une nouvelle forme de spécification axiomatique, qui sera dorénavant conditionnelle :

$$\text{si } \overline{p(t_1)} \text{ alors } f(\overline{cons}) == t_2$$

avec t_1 sous les mêmes restrictions que t_2.

Ainsi, le prédicat p_1 de la forme clausale correspondra à cette précondition p.

$$==(\overline{f(cons,X)},t_2) \Leftarrow \overline{p_1(t_1)}$$

D. Etape n°2: introduction de la sémantique procédurale

L'idée principale de la programmation logique (David WARREN, in "The art of Prolog" de L.STERLING et L.SHAPIRO (1986)) est que la déduction peut être vue comme une forme procédurale, et qu'une déclaration de la forme :

"P if Q and R and S."

peut également être interprétée procéduralement comme :

Pour résoudre P, il faut résoudre Q et R et S.

Quant à la programmation fonctionnelle, elle exprime la même déclaration par :

Si l'exécution de Q et R et S donne un résultat,
Alors, l'exécution de P donne ce même résultat

Selon la signification de la relation d'équivalence sémantique '==' (vue au chapitre deux) nous savons que le résultat de l'application de la fonction f est équivalente au résultat de l'application de t.

Dans cette optique procédurale et fonctionnelle, nous allons définir un nouveau prédicat appelé 'eval' qui sera le seul à être utilisé par le client du module comprenant les procédures Prolog d'un TDA.

par définition (et suite à la notion informelle) :

$$==(f(cons,X),t) \equiv_{def} R = evaluer(f(cons,X)) \Leftrightarrow R = evaluer(t)$$

Afin de manipuler aisément une méta-fonction d'évaluation d'un terme pour fournir le résultat, toujours dans une vue procédurale et fonctionnelle, nous avons pris comme *convention* que eval serait une procédure relationnelle ayant le résultat comme dernier élément de la liste des arguments de la fonction.

La sémantique de eval est :

$$\textbf{eval(t,R)} \Leftrightarrow \textbf{R = evaluer(t)}$$

Nous écrirons donc notre définition fonctionnelle d'un axiome équationnel par l'implication logique suivante :

$$\textbf{eval(f(cons,X),R)} \Leftrightarrow \textbf{eval(t,R)}$$

La nécessité de la transformation de '==' en 'eval' est due aux contraintes procédurales liées à la signification de '=='. En effet, le souci du client est

- soit d'exécuter une fonction sur des arguments dont on connaît la valeur, et d'obtenir un résultat (*optique application*),

  in = ground cons, X , **var** R
  out = ground cons, X , R

- soit de vérifier si une fonction dont les arguments sont connus fournira bien un résultat également connu (*optique vérification*).

  in = ground cons, X , R
  out = ground cons, X , R

Comme le client ne s'intéresse qu'à l'application de la fonction f, et non pas à celle du terme de droite t, nous transformons l'équivalence logique entre les deux 'eval' en une implication logique droite-gauche :

$$\textbf{eval(f(cons,X),R)} \Leftarrow \textbf{eval(t,R)}$$

Si une précondition détermine l'application de la fonction, nous placons celle-ci avant l'évaluation de t. Ce n'est donc plus le 'et' logique (**&**), mais le 'et' procédural (@) qui est employé :

$$\textbf{(eval(f(cons,X),R)} \Leftarrow \textbf{eval(t,R))} \Leftarrow \textbf{p}(t_1)$$

Or, comme $(A \Leftarrow B) \Leftarrow C \equiv A \Leftarrow B \,\&\, C$, nous obtenons :

$$\mathbf{eval(f(cons,X),R) \Leftarrow eval(t,R) @ p(t_1)}$$

$$\mathbf{eval(f(cons,X),R) \Leftarrow p(t_1) @ eval(t,R)}$$

1. Remarque

EHRIG et MAHR (1985) ont défini récursivement l'évaluation de fonction. Voici cette définition, réécrite dans notre syntaxe :

Soit TERME l'ensemble des termes de la signature SIG
A une algèbre de SIG,
Alors,

a) $eval(f)=f_A, \quad \forall f \in$ CONSTRUCT

b) $eval(f(t_1,...,t_n))=f_A(eval(t_1),...,eval(t_n))$

$\forall f(t_1,...,t_n) \in$ TERME

En pratique, cela signifie que pour évaluer le résultat d'une fonction, il faut d'abord évaluer les résultats de l'application des arguments, et ainsi de suite, jusqu'à ce que tous les arguments soient des constructeurs de base 0-aire de TDAs. Ensuite, on pourra reconstruire le résultat par application de ces constructeurs.

2. Généralisation du concept d'évaluation

Dans la définition récursive de EHRIG et MAHR, rien n'interdit que les termes soient des foncteurs n-aires. Or, jusqu'à présent, nous n'avions considéré que des constructeurs de base et des variables.

Afin d'étendre la notion d'évaluation et de l'ajuster à son sens réel, tous les arguments seront des variables, et devront être évalués à leur tour et comparés avec les constructeurs de base. Ensuite, s'il y a égalité syntaxique entre ces termes, nous pourrons évaluer le terme de droite t.

$$\mathbf{eval(f(cons,X),R) \Leftarrow eval(t,R)}$$
$$\mathbf{eval(f(Y),R) \Leftarrow Y=cons @ eval(t,R)}$$
$$\mathbf{eval(f(Y),R) \Leftarrow eval(Y,Y_1) @ Y_1=cons @ eval(t,R)}$$

avec $Y \equiv cons \cup X$

Ainsi, les arguments du vecteur de variable Y pourront être des termes sans restrictions aucune.

En ce qui concerne les préconditions, si le prédicat correspondant porte sur des variables appartenant à Y, il sera vérifié après l'évaluation de ceux-ci. Sinon, il peut être intéressant de le placer avant. Dans ce dernier cas, le libre choix est laissé au spécifieur-transformateur.

$$\mathbf{eval(f(Y),R) \Leftarrow eval(Y,Y_1) \ @ \ Y_1=cons \ @ \ p(t_1) \ @ \ eval(t,R)}$$

ou

$$\mathbf{eval(f(Y),R) \Leftarrow p(t_1) \ @ \ eval(Y,Y_1) \ @ \ Y_1=cons \ @ \ eval(t,R)}$$

E. Etape n°3: transformation systématique en Prolog

Afin de transformer les algorithmes logiques obtenus en procédures exécutables Prolog, il faut s'assurer que ceux-ci sont sous forme standard.

En vis-à-vis de cette page est exposée la liste des transformations basées sur les équivalences classiques en FOPL, extension de la liste présentée dans LLOYD (1984).

Une simple transformation finale permet d'obtenir des procédures logiques pures dérivées, dont les propriétés et l'exactitude sont discutées dans DEVILLE ('86) :

$$\mathbf{A \Leftarrow W_1 \ @ \ ... \ @ \ W_m \ \ devient \ \ A \leftarrow W_1,...,W_m.}$$

Enfin, quelques changements syntaxiques sont nécessaires pour obtenir des procédures Prolog.

- '←' en ':-'
- En langage Prolog, les variables qui ne sont présentes qu'une seule fois dans une clause peuvent être remplacées par la variable anonyme '_', ce qui permet une efficacité en gestion de mémoire, problème dû à l'aspect récursif des procédures.

F. Récapitulatif des formes transformées

axiomes TDAs (équationnels)

1. $f(cons)==t_2$ **(simple)**
2. si $p(t_1)$ alors $f(cons)==t_2$ **(conditionnelle)**
3. si $p(t_1)$ alors $f(cons,X)==t_2$ **(généralisée)**
4. si $p(t_1)$ alors $==(f(cons,X),t_2)$ **(préfixée)**

clauses FOPL (implication)

5. $==(f(cons,X),t_2) \Leftarrow p(t_1)$

clauses FOPL procédurales

6. $eval(f(cons,X),R) \Leftarrow p(t_1) @ eval(t_2,R)$ **(simple)**
7. $eval(f(Y),R) \Leftarrow p(t_1) @ Y=cons @ eval(t_2,R)$ **(variabilisée)**
8. $eval(f(Y),R) \Leftarrow p(t_1) @ eval(Y,Y_1) @ Y_1=cons @ eval(t_2,R)$ **(généralisée)**

procédures logiques pures dérivées

9. $eval(f(Y),R) \leftarrow p(t_1),eval(Y,Y_1),Y_1=cons,eval(t_2,R).$

procédures Prolog

10. $eval(f(Y),R)$:- $p(t_1),eval(Y,Y_1),Y_1=cons\_prolog,eval(t_2,R).$

G. Etape n°4 : choix d'implémentation des structures

Jusqu'à présent, abstraction complète a été faite des détails de représentation Prolog. En effet, bien que la plupart des lecteurs savent que la liste vide est [] et la liste à une élément de tête E et à une sous-liste queue L est [E|L], par contre, comment manipuler un produit cartésien ou une union? De plus, comment adapter nos axiomes fonctionnels de telle sorte que les constructeurs des TDAs aient leur équivalent en Prolog? Ce qui suit tente d'y répondre.

Selon Liskov et Berzins (76), l'implémentation doit avoir le même comportement que les spécifications, mais elle n'est pas contrainte à l'utilisation du même algorithme et de la même représentation pour réaliser ce comportement.

exemple : une implémentation d'une fonction manipulant le type tableau ne doit pas nécessairement retenir l'ancienne valeur de ce tableau.

En utilisant la programmation logique, on s'aperçoit que peu de problèmes de représentation des objets existent. En effet, Prolog n'offre que peu d'"instructions".

exemple :

t1 is t2 qui force l'évaluation de l'expression t2 pour obtenir t1.
conditions d'application : in(ground t2,var t1) : out(grd t2,grd t1)
in(ground t2,grd t1) : out(grd t2,grd t1)

[]	constructeur de liste vide,
[E|S]	constructeur de liste avec un élément de tête et une sous-liste formée par les éléments suivants,
0	constructeur du zéro,
t1 + t2	constructeur de l'addition.

La difficulté d'implémentation des types de données abstraits ne s'exprimera que dans le choix de représentation des constructeurs de base des TDAs. Tout est ramené, dans les axiomes spécifiant la sémantique des fonctions, aux constructeurs de base.

Prenons l'exemple du produit cartésien. La spécification précise pour un produit cartésien, les noms et les valeurs de chqaue champ le composant.

Au niveau représentation abstraite, un produit cartésien possède un constructeur de base qui est pc_cons. Celui-ci reçoit en argument le nom et la valeur initiale pour tous les champs. De là, trois autres fonctions ont été spécifiées, à savoir pc_modify, pc_sel et pc_is_in.

Au niveau implémentation, il faut décider de la représentation interne au langage de programmation d'un produit cartésien. Nous avons décidé que le produit cartésien serait une liste de couples "champ". Chaque couple est une liste de deux éléments : le nom et la valeur du champ.

```
eval(P,pc_cons(N1,V1,N2,V2))  >prolog>  eval(P, [ [N1,V1] , [N2,V2] ])
P=pc_sons(N1,V1,N2,V2)        >prolog>  P= [ [N1,V1] , [N2,V2] ]
```

Grâce à ces règles d'implémentation, il faudra remplacer, dans chaque clause, les parties de gauche des règles par les parties de droite.

H. Remarque sur le concept d'interface utilisateur

L'interface utilisateur est l'ensemble des informations que doit connaître celui qui désire employer des spécifications, ou des procédures exécutables. Ainsi, un concepteur de programme réutilise des spécifications et des procédures existantes réunies au sein d'une librairie, c'est-à-dire d'un ensemble non limité, extensible, et certifié correct. L'intérêt de ce procédé de programmation ou de spécification est évident :

- gain de temps, rapidité de création de nouveaux systèmes,
- certitude que les erreurs ne sont que dans les nouvelles spécifications apportées,
- standardisation des sytèmes par une uniformisation des spécifications (et de là, une compréhension facilitée).

Nous avons défini, dans la première partie, que l'interface utilisateur au niveau spécification est composée des cadres de spécification syntaxique des types de données abstraits.

Concernant la partie programmation logique avec l'objectif fonctionnel, l'interface utilisateur est l'ensemble des clauses Prolog de l'évaluateur puisque toutes les procédures appelées sont d'ordre "publiques". Les procédures privées (ou de travail) au module à programmer ne sont pas admises dans l'évaluateur.

TROISIEME PARTIE : APPLICATIONS DES METHODES THEORIQUES

Au stade actuel du travail, toutes les fondations théoriques utiles à la réalisation d'une méthode de conception de programme sont connues du lecteur. Mais cette base théorique se doit d'être enrichie par une expérimentation de la validité de la méthode dans certains cas concrets.

Partant de cette idée, nous nous sommes intéressés aux objets nécessaires à tout programme de gestion. Le choix de ces objets a été facilité par la présence de certains propriétés, à savoir :

- une généralité de concept,
- une capacité de structuration d'autres objets,
- une facilité toute relative d'abstraction et de conception.

En est sorti deux grands groupes d'objets simples, selon le second critère : les types d'objets paramétrés et les types d'objets non paramétrés. Tous les types de données autres que ceux qui font partie de ces groupes ne sont que des compositions ou des spécialisations de ceux-ci.

Chaque développement de types, depuis la spécification en TDAs, jusqu'à l'implémentation en procédures Prolog, s'est organisé en un module de conception dont le nom porte celui du type de données étudié. La liste des modules est parsemée dans le tableau suivant :

	SIMPLES	COMPOSES	SPECIALISES
NON PARAMETRES	Booléens Naturels Entiers Caractères	Date	Chaine de caractère
PARAMETRES	Séquence Union disjointe Produit cartésien	Table	Ensemble Pile

L'application des méthodes théoriques présentées dans les deux première parties a permis de conclure quelques remarques utiles pour leur raffinements ou leur adaptation éventuelle. Nous les présenterons, non pas par module, mais en généralisant leur conséquences, et en structurant ceux-ci en plusieurs points d'explicitation. Les références et exemples des moduels sont repris, pour faciliter la tache du lecteur.

Chaque point à éclaircir et/ou à détailler sera introduit par les questions que l'utilisateur des méthodes serait susceptible de se poser. Quelques exemples repris des annexes illustrerons les problèmes et leur résolution.

CHAPITRE 6 : Application de la méthode de spécification

Afin d'assurer à la méthode théorique de ce travail une évaluation de ses caractéristiques et possibilités, nous l'avons essayée sur plusieurs types de données, de préférence utiles à tous les programmes. La réalisation pratique des spécifications et transformations en Prolog forme l'annexe du mémoire. En voici les problèmes et solutions ou explications.

A.Choix du procédé de spécification d'un type de données

"Quand faut-il utiliser la méthode de spécification que préconise ce travail ?"

Tout dépend du type des données à spécifier. Dans le cas d'objets aisément compréhensibles à l'aide des fonctions les caractérisant, sans réutilisation ou notion d'héritage d'autres types, cette méthode est idéale : elle permet de construire rapidement une librairie de spécifications abstraites correctes.

Par contre, en ce qui concerne les objets complexes à concevoir en quelques fonctions, une méthode de décomposition en objets plus simples, avec explicitation des relations entre eux, pourrait être tout aussi approprié. *RSL, Requirements Specification Language* (Dubois ('84), Langelez-Paris ('86), Demeulemeester-Laloy ('89)) est un exemple de langage d'explicitation de relations complexes, en élaborant des procédés de simplification jusqu'à l'obtention de *types de données élémentaires* (En RSL, on préfèrera parler d'*opérations terminales*).

Déf. Un type de donnée élémentaire correspond à un ensemble d'objets appartenant à un même ensemble, dont :

- soit la perception abstraite est immédiate (simple et globale),
- soit nous disposons déjà de la spécification de ce type au sein d'une librairie.

Selon cette définition, les objets complexes deviennent élémentaires dès que leur spécification est écrite et insérée dans la librairie (avec, bien entendu, l'étape de validation terminée).

Les types de données présentés dans ce mémoire sont de ces deux classes (simples ou complexes). Toutefois, seule notre démarche a été appliquée, avec succès. Cependant, la possibilité d'une explicitation RSL, plus naturelle, a été décrite dans certains modules.

exemple :

Le module TABLE peut être vu comme une séquence d'éléments, dont on peut réutiliser les axiomes, avec des contraintes supplémentaires (assertions) :

sur la position des éléments de la séquence (borne inférieure et borne supérieure),
sur la longueur de la séquence (fixe).

TABLE[ELEM,NAT,NAT] |-> PC[low:NAT, high:NAT, liste:SEQ[ELEM]]

Ici, la table est vue comme un produit cartésien de trois éléments : une borne inférieure, une borne supérieure, et la liste des éléments de type ELEM.

Soit la fonction *bottom* qui fournit la valeur de la borne inférieure.

TDA : bottom(alloc(E,N1,N2)) == N1
bottom(store(E,N,T)) == bottom(T)

RSL : bottom(T) == pc_sel(low,T)

Plusieurs noms caractérisent cette façon de spécifier : explicitation, réutilisation des connaissances, spécification par abstraction, spécialisation, héritage. Bien sûr, les nuances existent et dépendent souvent de l'optique dans laquelle ils sont utilisés, mais ce n'est pas le sujet de ce mémoire.

Liskov et Berzins (1976) ont récapitulé les distinctions existant entre spécification axiomatique (TDA, p.ex.) et spécification par abstraction (RSL, p.ex.) :

- La méthode par abstraction est plus facile à comprendre et à construire pour les programmeurs, de par sa ressemblance avec un programme. Cependant, elle à tendance à fournir des informations détaillées qui ne font pas partie de l'abstraction à proprement parler.

- La méthode axiomatique est conseillée pour les preuves de programme. Quant aux preuves d'implémentation, elles sont probablement aussi difficile dans les deux approches.

Remarquons qu'il est parfois difficile de distinguer où commence l'implémentation d'un type spécifié par abstraction, puisque le fait de le décrire en terme d'un autre est déjà une façon de l'implémenter.

Voici la liste des avantages et désavantages de la méthode de réutilisation des connaissances (ou abstraction) que nous avons trouvés :

Avantages :

- Elle rattache les concepts à des concepts déjà connus,
- Elle simplifie l'écriture des spécifications (moins lourd, plus lisible),
- Elle améliore les performances (limite le nombre d'axiomes),
- Elle ne nécessite pas de connaissances des outils de notre travail.

Désavantages :

- La clarté n'est pas évidente si on ne manipule pas les types réutilisés,
- Le risque d'erreurs n'est pas nul :
  . Il faut être certain de l'exactitude de ce que l'on réutilise,
  . Il faut comprendre et réutliser à bon escient.

B. Choix des fonctions caractérisant un type

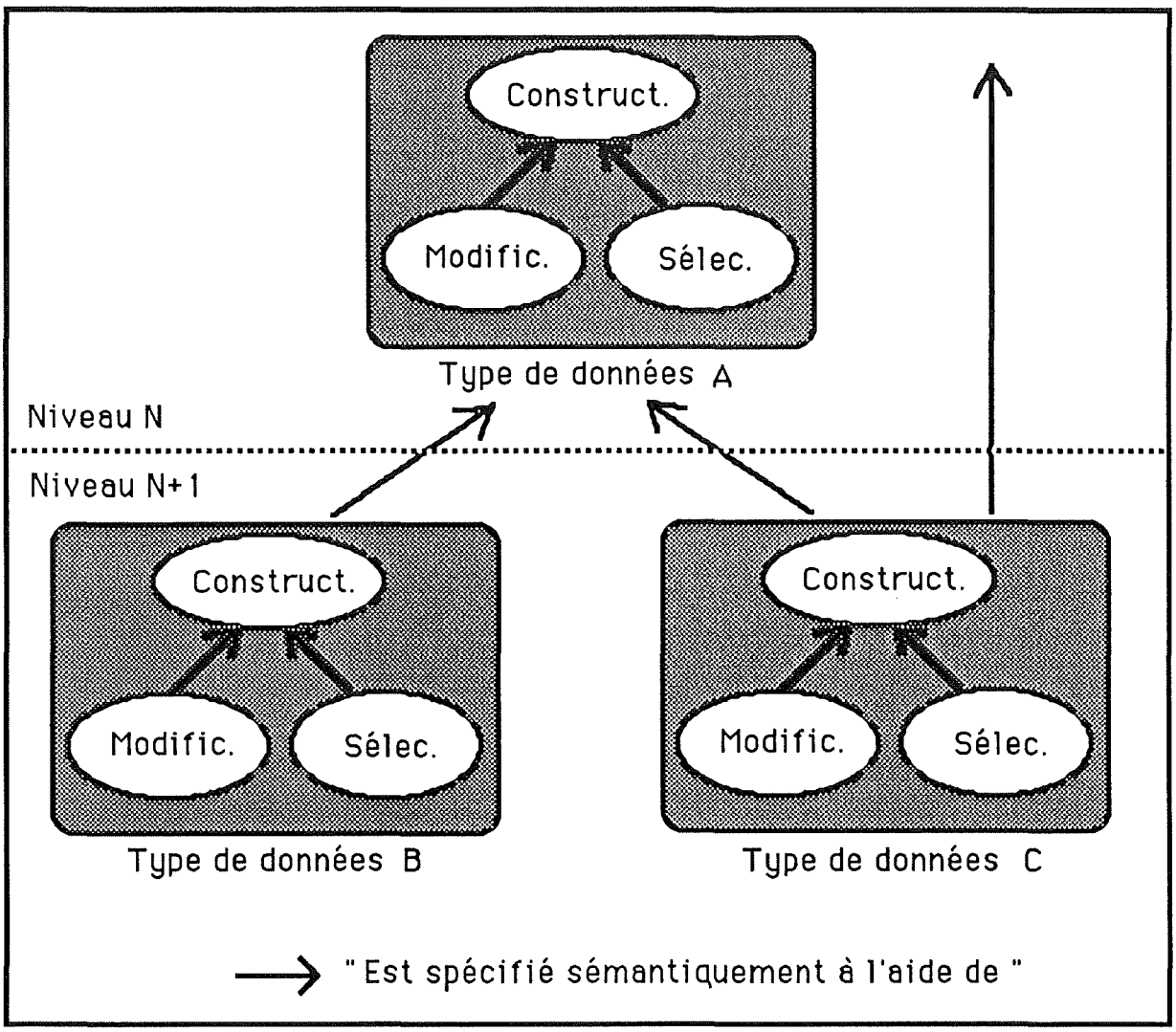
"Quelles sont les fonctions qui caractérisent un type ?"
"Comment les trouver ?"
"Comment les expliciter ?"

Bien heureux qui pourrait d'un seul coup, décrire toutes les fonctions possibles et imaginables d'un type. Par définition, un module reste *semi-ouvert* .

Déf . Un module est *ouvert* quant il accepte de recevoir de nouvelles propriétés (en terme de nouvelles fonctions), ou qu'il accepte que les propriétés existantes soient revues.

Nous précisons semi-ouvert car les constructeurs de base doivent être défini une fois pour toute et ne peuvent être modifiés sans changer le sens même du type de données. Ils forment le squelette, l'essence d'un type abstrait. Par contre, l'environnement d'un type abstrait (l'ensemble des modificateurs et des sélecteurs) est mouvant, extensible, éventuellement rétractable. Toutefois, il est dangereux de supprimer des fonctions d'un module si celui-ci est réutilisé par d'autres.

Cette notion de réutilisation crée une hiérarchie de spécification définie par la relation "est spécifié à l'aide des fonctions de". Un schéma global de représentation d'une telle architecture est donné ci-dessous.



- Une hiérarchie de spécifications -

En conclusion, l'effort important de conception à fournir se situe primordialement autour des constructeurs de base. Toute spécification axiomatique équationnelle ramène un type de données à une composition de constructeurs primordiaux. Appelons-la le *modèle de représentation abstrait* d'un objet d'un type donné.

exemple :

l'objet mathématique réel est représenté abstraitement par

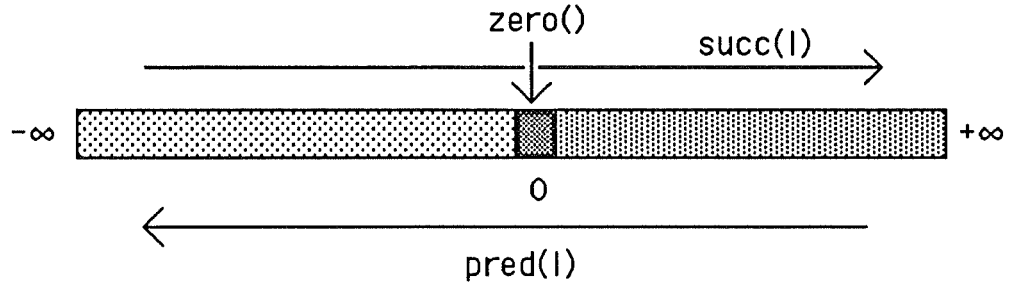
5	succ(succ(succ(succ(succ(zero())))))
length([5,4])	succ(succ(zero()))
[5,4,3]	app(5,app(4,app(3,empty())))

Nous nous sommes trouvé face à un problème de ce type. Comment représenter de façon adéquate les entiers ?. Alors que le problème est trivial pour les naturels, un dilemme est apparu pour le type ENTIER.

Premier choix de représentation

zero : --> INT
succ : INT --> INT
pred : INT --> INT

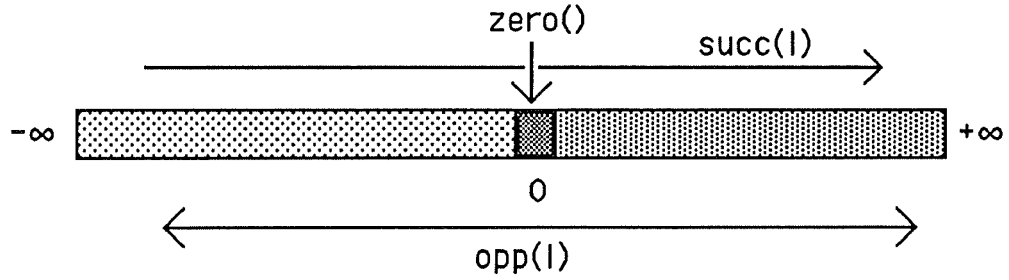
Un entier est nul, le successeur d'un entier, ou le prédécesseur d'un entier.



Second choix de représentation

zero : --> INT
succ : INT --> INT
opp : INT --> INT

Un entier est nul, le successeur d'un entier, ou la valeur négative d'un entier.



Dans les deux cas, les spécifications syntaxiques sont identiques. De plus, deux constructeurs sont les mêmes : zero() et succ(INT). Ils correspondent en fait aux constructeurs des NATURELS.

Alors, sont-ce des spécifications différentes, et si oui, qu'elle est celle qu'il faut retenir ? Au cours de l'écriture des axiomes de certaines fonctions, aucune différence n'est apparue, sauf lorsqu'il fallut résoudre les fonctions qui sont constructeurs de base dans l'autre spécification.

Premier choix de représentation

opp(zero()) == zero()
opp(succ(I)) == ...

Cette spécification n'a pas pu être résolue. Est-ce à dire qu'elle n'existe pas ? Non, disons plutôt qu'elle risque d'être trop complexe, et ainsi, peu naturelle. Voyons le second cas.

Second choix de représentation

pred(zero()) ==opp(succ(zero()))
pred(succ(I)) == I
pred(opp(I)) == opp(succ(I))

Ici, aucun problème de représentation.

Donc, si l'on s'aperçoit qu'il est délicat ou irréalisable de définir de nouvelles fonctions à l'aide des constructeurs de base, il est conseillé de revoir fondamentalement ceux-ci. Bien entendu, la réalisation la plus complète possible des spécifications sémantiques du type est idéale pour s'apercevoir très tôt que les constructeurs de base sont les bons.

Une fois que les constructeurs de base sont définis, choisir les autres fonctions n'est que question d'utilisation du type. Il suffit de se demander ce que l'on veut faire du type pour trouver les fonctions qui le caractérisent.

Deux modes de pensée sont possible lors de la construction de l'ensemble des fonctions :

- Une *vue à court terme*, réalisée par une *conception en largeur* :
"J'ai besoin de connaître la longueur d'une séquence pour résoudre mon problème."
Une fonction n'est à créer que dans la mesure où elle est nécessaire. Son intérêt est lié à sa nécessité.

- Une *vue à long terme*, réalisée par une *conception en profondeur* :
"Si je dispose d'une séquence, il pourrait être intéressant d'en connaître sa longueur"
Une fonction est crée dans la mesure où elle est pourrait être nécessaire. Son intérêt existe par défaut. Son utilisation est liée à sa nécessité.

C. Elimination de la redondance de spécifications

"Malgré l'assurance que mes spécifications sont complètes et cohérentes, ne risque-t'il pas d'y avoir surspécification par redondance de certains axiomes ?"
"Comment éliminer la redondance si elle existe ?"

Un des gros désavantages de certaines spécifications est la présence de redondances, surspécifiant et alourdissant la sémantique. En effet, bien souvent, la même choses est dite plusieurs fois sous des formes différentes. Deux cas se présentent à nous pour résoudre ce désavantage :

- Les axiomes sont construits systématiquement, en envisageant tous les cas de figure possibles et imaginables (c'est-à-dire, autant d'axiomes que de combinaisons des constructeurs de base de chaque paramètre).

- Postulons que la redondance ne peut apparaître si l'idée de départ est suffisamment abstraite. Dans ce cas, la démarche de construction par induction structurelle offrira un ensemble d'axiomes non redondants. Toutefois, il faut être certain qu'aucun cas n'est oublié. L'élimination de la redondance peut provoquer une perte de la complétude.

exemple :

VUE N°1 : Combinaisons exhaustives des constructeurs

```
and(true(),true())     == true()
and(true(),false())    == false()
and(false(),true())    == false()
and(false(),false())   == false()
```

VUE N°2 : On peut observer que si false() est présent dans les arguments, le résultat de la fonction est false().

```
and(true(),true())   == true()
and(false(),B)       == false()
and(B,false())       == false()
```

VUE N°3 : Une autre observation valide est de constater que la fonction donne la valeur logique true() que si ses deux arguments sont true().

```
and(true(),true())   == true()
and(B1,B2)           == false()
```

Apparaît un problème de taille dans la spécification axiomatique : l'*ordonnancement des axiomes.* Dans la troisième vue, le deuxième axiome est la réfutation du cas précédents. En effet, si le premier n'est pas celui que l'on souhaite, il faut essayer le second. Si l'ordre inverse est considéré, on s'aperçoit que l'on aura une erreur, car dans tous les cas, on pourra choisir le second axiome (and(B1,B2)).

Ainsi, le mécanisme de la perception de la réalité influe sur la façon de la spécifier, et par là, d'obtenir une spécification correcte.

Si les spécifications correspondent à un ordonnancement des axiomes, celui-ci DOIT être préciser dans les spécifications. Afin d'éviter cette précision, nous avons décidé que toutes nos spécifications sont implicitement ordonnancés de haut en bas. Les derniers ne peuvent être vérifiés que si les premiers n'ont pas à l'être ou ne l'ont pas été. Il faudra en tenir compte dans la méthode de transformation, et dans le choix du langage d'implémentation.

D. La spécification des erreurs et du comportement du syst dans ce cas

"Comment spécifier qu'une combinaison de constructeurs pour un axiome est impossible ?"
"Comment spécifier un système qui résoud les cas d'erreurs ?"

Normallement, les axiomes qui sont impossibles pour une combinaison de constructeurs ne devraient pas être présents dans la spécification. Cette option est de dire que ce qui n'est pas décrit n'a pas à se produire.

Mais il est pensable que l'utilisation de tels axiomes provoque une erreur du système, ou que le concepteur veuille s'assurer que l'utilisateur emploiera les axiomes correctement. Pour satisfaire ces besoins, plusieurs options sont proposées :

- Le résultat d'une fonction n'est pas le seul. On ajoute un résultat implicite complémentaire qui prend une valeur fixe (0 par exemple) si tout va bien, ou une valeur dépendante du type d'erreur (n° ou intitulé) en cas d'erreur. Ceci doit apparaître dans les spécifications (option non retenue pour ce travail).

- Le résultat d'une fonction est unique, mais en cas d'erreur, la fonction appelle une autre fonction dont le seul but est d'assurer l'identification de l'erreur, l'avertissement de l'utilisateur, et l'éventuelle reprise sur erreur. Cette fonction est prédicat

extra-logique qui exécute une fonction système ERREUR avec un paramètre. Elle est système-dépendant et ne sera conçu que de façon complète qu'à l'implémentation. Les arguments de cette fonction doivent préciser les fonctionnalités précédentes.

- On spécifie le mode d'utilisation des fonctions grâce à des assertions de deux types : les préconditions et les post-conditions. Dès lors, seul l'utilisateur est responsable de l'utilisation des fonctions, puisqu'il sait par l'interface que certaines conditions doivent être remplies avant et après (option retenue dans ce travail).

La conséquence du choix que l'on a fait est que, lors de la construction par induction structurelle, certaines formes des paramètres d'induction seront éliminées. De plus cela simplifie la lecture des axiomes.

exemple :

1. is_at(E,zero(),S) == ERREUR "Valeur de paramètre fausse"
   ou ERREUR "Indice incorrect"

2. FONCTION is_at(E,N,S) DONNE Br.
   PRECONDITIONS : N >= succ(zero()), N <= length(S)

exemple :

1. del(N,empty()) == ERROR ou UNDEFINED ou empty() ?

2. PRECONDITIONS : not(is_empty(S))

exemple :

1. pos(E,empty()) == ERREUR "Il n'y a pas d'éléments dans une liste vide" ou zero()

2. PRECONDITIONS : not(is_empty(S))

E. La spécification des cas non définis et du comportement système dans ce cas

"Comment spécifier, dans les axiomes, qu'une valeur de résultat est indéfinie, ou correspond à un domaine de valeurs ?"

Les cas non définis (Undefined) sont dû à trois raisons :

- Ils correspondent parfois à un manque de spécifications ou ,
- Dans l'état actuel des connaissances, il n'y a pas moyen de donner une valeur au résultat, ou encore,
- Le résultat est un ensemble de valeurs (variable) et délimite, dès lors, le domaine des valeurs possibles.

exemple :

del(N,empty()) == UNDEFINED ou empty() ?

div(I,zero())== UNDEFINED (domaine de valeur infini ou indéterminé)

Nous pouvons faire les mêmes remarques et proposer les mêmes solutions que pour les traitement d'erreurs. Inutile de s'y attarder.

F. L'ordre des spécifications

"L'ordre des axiomes a-t'il une importance ?"
"Comment ordonner les axiomes ?"

Ceci a déjà été présenté au point C. L'ordre des axiomes est imposé par l'optique de réfutation des cas précédents. Bien entendu, en théorie des spécifications axiomatiques, l'ordre des axiomes est indéfini. Voir un ordre dans les axiomes est du même moule que d'instancier le moteur d'inférence logique en Prolog. C'est un choix "d'implémentation" afin d'améliorer les performances, ou de faciliter le travail. Ici, ce choix simplifie l'écriture et la lecture des règles. Dans ce mémoire, il est implicite que les axiomes les derniers seront utilisés que si les précédents ne le sont pas.

G. Les constructeurs de base sont des fonctions systèmes

"Comment se fait-il qu'il ne faille point spécifier la sémantique des constructeurs primordiaux ?"

Les Types de données abstraits sont définis par leurs constructeurs de base. Ceci contourne, ou retarde, le problème de représentation matérielle des objets. Ainsi, la sémantique des constructeurs est implicite. Elle est explicitée en français, mais ne sera réellement construite qu'à l'implémentation dans un langage précis. Ceci est du domaine du chapitre suivant.

H. Type du résultat

"Si le résultat est implicite à une fonction, on ne pourra jamais spécifier de procédures qui modifient la valeur d'un argument ?"

La distinction que certains voudraient faire entre les notions de procédure et de fonction telles que Pascal les définit n'a pas à apparaître au niveau des spécifications. Une opération abstraite EST une FONCTION et ce n'est qu'à l'implémentation que l'on décidera si celle-ci doit être une procédure. Ce choix dépendra certainement du type de la fonction. Si il s'agit d'un modificateur, il y aura de grandes chances que la fonction devienne une procédure qui modifiera l'argument de la sorte d'intérêt.

I. Equivalence et exactitude des spécifications

"Comment être certain que les axiomes sont corrects ?"

La certitude de l'exactitude des axiomes est assurée par la méthode de construction par induction structurelle. Deville ('87) a démontré mathématiquement que la méthode d'induction engendrait la complétude et la cohérence des axiomes. Bien sûr, un raisonnement valable est nécessaire lors des choix des paramètres d'induction et de leurs formes structurelles.

Voici deux axiomes équivalents, mais dont un est inexact (le second). En effet, bien que la forme structurelle du second argument soit plus petite (S2 < app(E,S2)), il n'en est pas de même pour N et pred(N). Pred(N) est une fonction de modification, et pas un constructeur de base. Ceci ne correspond pas à ce que permet la méthode de construction.

exemple :

```
del(succ(N),app(E,S2)) == app(E,del(N,S2))    <--
del(N,app(E,S2)) == app(E,del(pred(N)),S2)    <--
```

De ce fait, bien que deux asiomes soient équivalents, nous n'assurons leur exactitude que si ils sont construits par décomposition ou par induciton structurelle.

J. La formulation logique d'axiomes d'ordre supérieur à 1

"Puis-je spécifier des fonctions utilisant d'autres opérations en arguments ?"

Certaines opérations utilisent, comme argument, d'autres opérations (prédicats).

exemple : fonction FILTER (Seq, Pre) --> Seq

Le rôle de cette fonction est de filtrer une séquence selon une propriété donnée à vérifiée. cette propriété est un prédicat qui doit être vérifié pour les éléments de la séquence.

La formulation logique des axiomes de telles fonctions est d'ordre supérieur à 1 (selon le degré d'indentation de ce mécanisme). Le problème est qu'il est impossible d'exprimer logiquement ce type d'axiome, puisque les axiomes sont spécifiés sémantiquement à l'aide d'un langage des prédicats du premier ordre.

Nous ne connaissons pas de moyen théorique permettant de contourner ce problème. En fait, nous l'avons reporté jusqu'au moment de l'implémentation, en espérant que certaines caractéristiques extra-logiques nous permettraient de le résoudre.

exemple :

Soit une personne identifiée par son nom, son prénom et son âge, on a en spécification RSL :

```
PERSONNE   |-> CP [NOM,PRENOM,AGE]

NOM        |-> SEQ[CHAR]
PRENOM     |-> SEQ[CHAR]
AGE        |-> NATURAL
```

où CP, SEQ, CHAR et NATURAL sont des objets de la librairie.

En ayant une liste de personnes et de leur caractéristiques (nom, prénom et âge), on voudrait sélectionner la liste de toutes celles qui ont un âge supérieur à 25 ans.

filter (Liste_personne, >(age(A,x),25)).

Les spécifications axiomatiques de la fonction de sélection sur un prédicat ont pu être écrite :

filter(empty(),P) == empty()
filter(app(E,S),P) == p(E) & app(E,filter(S,P))
filter(app(E,S),P) == not(p(E)) & app(E,filter(S,P))

Ceci n'est évidemment pas cohérent avec la théorie du premier ordre, puisque notre idée est que la variable P sera instanciée par le prédicat p. Mais cela n'est pas grave si la spécification garde trace de cette explication.

En réalité, selon la limite de nos outils, la seule solution est d'écrire plusieurs fonctions de filtrage dont chacune concerne un prédicat défini.

exemple : sélection des éléments d'une séquence de produits cartésiens dont l'age est plus grand qu'un age limite donné.

if greater(pc_sel(E,age),Age_limite) then
filter_age(app(E,S),Age_limite) == app(E,filter_age(S,Age_limite))

Un problème identique qui s'est présenté dans ce travail est de spécifier une fonction qui vérifie le type de ses arguments.

exemple :

Soit *is_list* une fonction qui détermine si son argument est une liste ou non.

Le problème qui se pose ici est de savoir où classer cette fonction, de déterminer à quel type elle appartient. La seule explication à nos yeux est que, puisque les TDAs doivent avoir des arguments typés, on pourrait déterminé le type de l'argument comme étant un type d'ordre supérieur à 1, possédant les caractéristiques de tous les types possibles *(domaine des types*). Ainsi, la spécification de l'axiome de cette fonction doit être exprimée à l'aide de tous les constructeurs de base appartenant aux types du domaine des types.

exemple :

TYPE : TOUT_TYPE
DOMAINE DES TYPES : [BOOLEEN, NATUREL, SEQ]
FONCTION : is_list : TOUT_TYPE --> BOOLEAN

AXIOMES :

is_list(true()) == false()
is_list(false()) == false()
is_list(zero()) == false()
is_list(succ(N)) == false()
is_list(empty()) == true()
is_list(app(E,S)) == true()

Donc, la spécification axiomatique est complète et cohérente. Un élément de l'ensemble de ce type est construit à l'aide des constructeurs de base des types de son domaine des types, et la fonction is_list est construite à l'aide de ces constructeurs.

Bien entendu, dans la plupart des cas, le domaine des types reste ouvert à toute extension à d'autres types.

Ce mécanisme de spécifications pourrait également servir à la définition des types qui possèdent toutes les caractéristiques d'un type plus d'autres, originales. D'ailleurs, ceci a été fait intuitivement, et peut-être par hasard pour la spécification des entiers : les constructeurs de base des naturels sont présents, plus un nouveau constructeur permettant d'obtenir les entiers strictement négatifs.

K. La construction par induction : en parallèle ou en série

La méthode de construction par induction structurelle postule qu'il faut choisir un paramètre d'induction parmi les arguments de la fonction à spécifier. cependant, il arrive couramment qu'une seule induction de forme ne suffise pas. Il faut alors choisir parmi les arguments restants un nouveau paramètre d'induction, afin de simplifier le problème. Donc, la combinaison des paramètres d'induction est importante.

Ceci est l'induction classique que nous qualifierons d'*induction multiple en série.*

exemple : subseq(N,N2,S)
P.I. : N cons1 : zero()
cons2 : succ(N3)
$P.I._2$: N2 (seulement si N est succ(N3))

Mais un autre phénomène peut se produire. Supposons que nous ayons une fonction *last* dont le rôle est de fournir le dernier élément 'une liste. Pour la spécifier, nous choisissons comme paramètre d'induction le seul argument de la fonction, à savoir S.

cons1 : empty()
cons2 : app(E,S2)

- Si une liste est vide, elle n'a pas d'élément terminal. ceci nous rapporte aux problèmes des valeurs indéfinies ou des cas d'erreurs.

- Si une liste possède au moins un élément, le dernier élément de la liste est celui-là *Si* la queue de la liste est vide.

Nous sommes en présence d'une autre forme de combinaison d'induction, l'*induction simple en série.*

exemple :

last(app(E,empty()) == E
last(app(E,app(E2,S3)) == last(app(E2,S3))

ou encore

if S2=empty() then last(app(E,S2)) == E
if S2=app(E2,S3) then last(app(E,S2)) == last(S2)

Une troisième forme d'induction (outre la forme normale d'*induction simple normale*) est l'*induction multiple en parallèle* : afin de spécifier une opération, il faut nécessairement envisager deux ou plusieurs paramètres d'induction en même temps.

exemple :

P.I.s : S et S1

is_prefix(empty(),S2) == true()
is_prefix(S,empty()) == false()
is_prefix(app(E1,S1),app(E1,S3)) == is_prefix(S1,S3)
is_prefix(app(E1,S1),app(E3,S3)) == false()

CHAPITRE 7 : Application de la méthode de programmation

A. L'insertion d'assertions

Entière liberté est laissée, dans notre démarche, à l'utilisateur pour le bon usage des procédures. Ceci signifie que c'est l'utilisateur qui doit vérifier les préconditions sur les arguments. Voici un exemple dont l'utilisateur dispose pour insérer ces préconditions dans l'évaluateur.

Exemple :

```
préconditions : N>0 et N<=length(S)

if  and(greater(N,zero()),leq(N,length(S))
then eval(del(N,S),Sr) == def

eval(del(N,S),Sr)  :-  eval(N,Nr1),eval(S,Sr1),  eval(lengh(Sr1),L),
                          eval(and(greater(Nr1,zero()),leq(Nr1,L)),true),
                       ! , p_del(Nr1,Sr1,Sr).
```

B. Ordonnancement des clauses

Les clauses logique d'une même procédure n'ont, à priori, aucun ordre dans leur utilisation. Par contre, Prolog en impose un. Il appliquera d'abord les clauses qui sont en tête de texte, si celles-ci s'unifient avec la question. Sachant cela, il était intéressant d'utiliser cette caractéristique. C'est ce que nous avons fait pour certaines fonctions, et cela dès la spécification. Dans les spécifications, nous avons défini un ordre identique de priorité des axiomes, les derniers n'étant susceptibles d'être appliqués que si les précédents ne s'unifiaient pas.

C. L'utilisation de prédicats extra-logiques

"Puis-je programmer en Prolog des fonctions utilisant d'autres opérations en arguments ?"

Nous avons dit, dans le chapitre six, que nous ne connaissions pas de moyen théorique permettant d'exprimer logiquement des axiomes de fonctions dont la formulation logique est d'ordre supérieur à 1. En fait, nous avons reporté ce problème jusqu'à ce chapitre, en espérant que certaines caractéristiques extra-logiques nous permettraient de le résoudre. reprenons l'exemple du chapitre 6.

Soit la fonction FILTER (Seq_1, Pre) --> Seq_res

Le rôle de cette fonction est de filtrer une séquence Seq_1 pour obtenir une séquence Séq_res dont les éléments satisfont à une propriété donnée. Cette propriété est un prédicat qui doit être vérifié pour les éléments de la séquence.

exemple :

En ayant une liste de personnes et de leur caractéristiques (nom, prénom et âge), on voudrait sélectionner la liste de toutes celles qui ont un âge supérieur à 25 ans.

filter (Liste_personne, >(age(x,A),25)).

Il faut trouver un moyen de vérifier que chaque élément de la suite satisfait la propriété. Pour cela, nous faisons appel à certains prédicats extra-logiques (functor et arg), repris dans les procédures substitute et subst_args, dont les explications suivent l'exemple.

exemple : Si Séq_1 est la suivante :

[[dupont,robert,27],[dupond,marc,56],[lajoie,jean,12],[toto,luc,45]]

De la liste, on prendra le premier élément : [Dupont,Robert,27]. On remplacera le x de >(age(x,A),25) par cet élément, et on appliquera ce prédicat. Selon le résultat, on choisira la deuxième ou la troisième clause filter. Il faut alors désinstancier le prédicat propriété, et recommencer avec l'élément suivant.

La procédure Substitute

L'objectif de la procédure substitute est d'effectuer le remplacement de chaque occurence d'un terme par un autre dans un troisième. La procédure a quatres arguments : les deux premiers spécifient ce qui va être substitué (l'ancien terme), et par quoi (le nouveau terme). Les deux derniers indiquent le terme dans lequel la substitution a lieu et ce terme après substitution.

exemple : substituer le terme x par le terme [Dupont,Robert,27] dans le prédicat >(age(x,A),25)

L'analyse de la procédure substitute fait apparaître une procédure auxiliaire : subst-args qui réalise la substitution au niveau d'une occurence (alors que substitue remplace chaque occurence. Cette façon de décomposer le problème répétitif en une suite d'application d'un problème simple est classique en programmation logique).

substitute (New, Old, Val) --> Newval

Cette fonction donne un terme Newval qui est le terme Val dans lequel on a remplacé chaque occurence du terme Old par le terme New.

conditions d'applications in(gr.,gr.,gr.,var) / out(gr.,gr.,gr.,gr.)

```
substitute( New, Old, Old, New ) :-  !.
substitute( New, Old, Val, Val ) :-  atomic( Val ), !.
substitute( New, Old, Val, Newval ) :-
        functor(Val,Fn,  N),
        functor(NewVal,Fn,N),
        subst_args(N, New, Old, Val, Newval ).
```

La procédure Subst_args

subst_args(N, New, Old, Val) --> Newval

Cette fonction donne un terme Newval qui est le terme Val dans lequel on a remplacé la Nième occurence du terme Old par le terme New.

conditions d'applications in(gr.,gr.,gr.,gr.,var)/out(gr.,gr.,gr.,gr.,gr.)

```
subst_args( 0,_,_,_,_ ) :- !.
subst_args( N, New, Old, Val, Newval )   :-
        arg(N, Val, Oldarg),
        arg(N, Newval, Newarg),
        substitute (new, Old, Oldarg, Newarg),
        N1 is N-1,
        subst_args(N1, New, Old, Val, Newval).
```

Les procédures extra-logiques functor et arg

Quelques mots d'explication sur les procédures extra-logiques Prolog utilisées dans substitute et subst_args.

Tout d'abord, la procédure Prolog functor(Term,F,Arity) qui est vraie si Term est un terme dont le principal foncteur a pour nom F et pour arité Arity. Rappelons que l'arité d'une fonction est le nombre d'arguments qu'elle a.

Quant à la procédure arg(N,Term,Arg), elle est vraie Arg est le Nième argument du terme Term.

La procédure p-filter

La procédure logique p-filter emploie la procédure substitute avec le prédicat correspondant à la propriété et la valeur de l'élément courant dans la suite que l'on désire filtrer. Subsitute forunira un nouveau prédicat qu'il mettra dans la variable logique Goal. Pour effectuer la vérification de la propriété, il faut évaluer le prédicat de la variable Goal, dans lequel les occurences seront les occurences contenues dans l'élément de la liste actuelle au moment de l'appel à ce prédicat.

Du fait de l'utilisation de ces procédures "extra-logiques" (par indirection), il est nécessaire de faire apparaître un nouvel argument dans le prédicat qui utilisera ces procédures. En effet, il ne faut pas oublier de préciser le terme qui devra être substituer. Par convention, on peut le fixer dans substitute, mais cela manquerait de souplesse.

filter(Seq_arg,Prop,Var) --> Seq_res

Cette fonction donne une séquence Seq_res qui est la suite des éléments de la séquence Seq_arg qui satisfont tous à la propriété Prop, si l'on a remplacé le terme Var de la propriété par ces éléments, pour la vérification.

conditions d'applications in(gr.,gr.,gr.,var)/out(gr.,gr.,gr.,gr.)

```
p_filter([],[],_,_).
p_filter([E1|Sres],[E1|Sarg],Pre,Var) :-
        substitute(E1,Var,Pre,Goal), Goal, !,
        p_filter(Sres,Sarg,Pre, Var).
p_filter(Sres,[E1|Sarg],Pre,Var) :-
        p_filter(Sres,Sarg,Pre, Var).
```

D. Passage de l'évaluateur fonctionnel en procédures logiques

Afin de conserver, pour certaines fonctions, à la fois une interface utilisateur qu'offre l'évaluateur, et un prédicat dont la forme est logique, nous nous sommes intéressés à la manière de transformer l'évaluateur.

exemple :

Pour la fonction de concaténation, conserver le prédicat d'évaluation eval(conc(...),...), et en extraire le prédicat logique correspondant p_conc(...).

eval(conc(S1,S2),S3) et *p_conc(S1,S2,S3)*

Précisons clairement que le conc est différent dans les deux procédures : conc de l'évaluateur est un foncteur, alors que conc de la procédure logique est un prédicat.

Nous sommes parvenus à cela très facilement. En premier lieu, l'évaluateur est réécrit en considérant que l'évaluation d'un foncteur est la suite de l'évaluation de chacun de ses arguments (qui peuvent être également des foncteurs) et de l'appel au prédicat correspondant.

exemple :

```
eval(conc(S1,S2),S3)  :- def.
```

devient

```
eval(conc(S1,S2),S3) :- eval(S1,Sr1), eval(S2,Sr2),
                        p_conc(Sr1,Sr2,S3).
p_conc(Sr1,Sr2,S3)  :- def_simplifiée.
```

où def est la conjonction de prédicats définissant l'algorithme de détermination du résultat S3 à partir des arguments S1 et S2

Ensuite, la partie définition de droite doit être simplifiée pour être reprise dans la partie droite du prédicat. Certaines étapes sont à suivre :

- Les évaluations de termes sont réduites à une égalité syntaxique (=) de terme, et peuvent donc être éliminées par remplacement des termes dans leur position respective.

- Les évaluations de fonctions sont réduite à l'appel au prédicat correspondant, avec la convention que le dernier argument est le résultat.

exemple :

```
eval(subcut(E,E2,S),) :- eval(S,).
eval(subcut(E,E2,S),[E|Sr]) :- eval(S,[E|S2]), eval(prefcut(E2,S2),Sr).
eval(subcut(E,E2,S),Sr) :- eval(S, [E3|S2]), eval(subcut(E,E2,S2),Sr).

  eval(subcut(E,E2,S),Sr) :- eval(E,Er1), eval(E2,Er2),eval(S,Sr1),
                             p_subcut(Er1,Er2,Sr1,Sr).
  p_subcut(E,E2,S),).
  p_subcut(E,E2, [E|S2]), [E|Sr] ) :- p_prefcut(E2,S2,Sr).
  p_subcut(E,E2, [E3|S2]),Sr) :- p_subcut(E,E2,S2,Sr).
```

E. L'évaluation terminale de constantes

Jusqu'à présent, une omission importante a été faite, à savoir la règle d'arrêt de l'évaluateur. Quand l'évaluateur ne doit-il plus évaluer? En réalité, l'évaluateur ne doit plus évaluer les constantes. Ainsi, si l'on sait que l'on a un terme constante ou variable ou même foncteur à arité nulle, il faut rendre ce terme tel quel. Le foncteur à arité nulle est par définition égal sémantiquement à une constante, puisqu'il crée un résultat à partir de rien).

La façon de réaliser cela est, soit d'écrire pour chaque terme de ce type une clause de l'évaluateur, soit de trouver un moyen de déterminer si un terme est à arité nulle. Il existe, ceci a été dit au point précédent, des prédicats extra-logiques qui satisfont à notre souhait (functor).

```
eval(X,X) :- functor(X,_,0).
```

F. Amélioration des performances des procédures exécutables

Ce travail n'a pas tenté de chercher ni d'approfondir les moyens d'améliorer les performances de la logique ou du moteur d'inférence Prolog. Certaines solutions existent, telle que l'utilisation du "cut" qui permet d'éviter des recherches inutiles.

Toutefois, une remarque qui diminuera grandement l'espace mémoire réservé aux variables, et la récursivité est grande consommatrice de cette ressource, est de remplacer les variables inutilisées dans les procédures par les *variables anonymes*. Ces variables n'ont pas de mémoire réservée. Elles sont écrites par un tiret souligné ('_').

exemple : or(true,_,true).
or(_,true,true).

Si un des arguments est vrai, alors le résultat de l'opération logique or est vrai

CONCLUSIONS

Ce mémoire s'est efforcé de concevoir et d'analyser une démarche de programmation dans le cadre plus vaste du développement global de méthodes de prototypage.

L'objectif semble être atteint. En effet, nous avons pu développé une méthode en deux étapes. La première, étape de spécification, s'est fondée sur le langage des types de données abstraits et sur la logique des prédicats du premier ordre, et s'est inspiré de travaux préalables de constructions correctes d'axiomes. Quant à la deuxième, étape d'implémentation en langage logique, elle a été réalisée en langage logique Prolog, et a permis de systématiser la transformation des axiomes spécifiés en clauses logiques.

L'intérêt du développement de telles méthodologie sest multiple. En voici une liste non exhaustive :

- Réutilisabilité des spécifications (principe de réutilisation des connaissances).
- Localisation de celles-ci (pour modification) en modules.
- Gain de temps appréciable dans l'étape de spécification (opérations de base prédéfinies).
- Réduction du risque d'introduction d'erreurs.

La démarche proposée possède certains avantages non négligeables que les concepteurs apprécieront :

- Elle fonde ses bases sur des théories mathématiques rigoureuses, les types de données abstraits et la logique des prédicats du premier ordre.

- Le mécanisme d'abstraction est poussé dans ses derniers retranchements, afin de retarder au maximum le moment du choix de représentations physiques des objets. Ceci augmente la puissance de la démarche, en facilitant la portabilité des spécifications.

- Ces outils mathématiques confèrent aux méthodes des

possibilités de contrôles formels de cohérence et de complétude.

- C'est une démarche de bout en bout, ce qui lui confère la qualité de la cohérence entre tous les outils utilisés.

- Elle propose une certaine perception de la réalité, orientée dans un but fonctionnel.

- C'est une méthode *à court terme* qui permet d'obtenir rapidement des procédures exécutables, depuis l'analyse du problème. Ces qualités de rapidité, d'exactitude et de certitudes sont essentielles dans le cadre du prototypage de systèmes.

- C'est une méthode *à long terme*, par son caractère de documentation claire et par son objectif de construction d'une librairie de spécifications et de procédures correspondantes réutilisables.

Mais toute méthode présente des aspects négatifs. Dans ce travail, ce sont les suivants :

- Sa jeunesse est certainement son plus gros péché. On ne peut savoir si la méthode tient la route sans l'avoir utilisée à long terme. Nous n'assurons la validité expérimentale que pour les types de données auxquels nous nous sommes intéressés, bien que la validité théorique existe.

- Le corollaire du premier désavantage est que ce travail n'a implémenté que des types de base. On connait encore mal le comportement de la démarche dans le cas de types complexes. Mais que le lecteur se rassure, ce travail précise qu'une autre méthode , d'un niveau supérieur, permet de simplifier les types complexes par explicitation R.S.L.

En conclusion, la démarche de conception de procédures fonctionnelles depuis un problème posé en terme de manipulation de données est satisfaisante et permet de réduire considérablement les coûts de conception et même de maintenance. En effet, cette démarche allie les aspects construction et documentation en un ensemble cohérent et compact, ce qui offre une maniabilité idéale.

Certaines suggestions concernant quelques extensions futures à ce mémoire sont décrits à la page suivante.

Il serait intéressant d'*automatiser* les différentes étapes de la démarche. Certainement que des difficultés inhérentes à la complexité du problème surgiront, mais celles-ci seront sans doute rapidement éliminées. Toutefois, les informations en entrée, ou plus exactement la méthode entière de spécification, ne pourront être complètement créés. L'automatisation de cette étape doit se faire dans l'optique d'une aide à la conception assistée par ordinateur.

En ce qui concerne l'étape de transformation, affirmons que sur base des résultats obtenus manuellement, aucun obstacle ne s'oppose à l'automatisation totale de cette étape, exceptée, bien entendu, la phase de détermination de l'implémentation des constructeurs de base, qui reste du domaine d'une certaine intelligence.

Vu la proximité de concepts entre cette démarche et la méthode de programmation orientée-objet, une analyse comparative détaillée et assurant le rapprochement des deux permettrait de tirer des conclusions intéressantes.

La validité toute théorique des procédures Prolog obtenues n'assure pas que le système réagisse comme on l'attend. Une étape de vérification des procédures exécutables, pour un rétro-contrôle, n'est pas à dédaigner.

Enfin, n'oublions pas que le rôle de ce travail est de s'intégrer parfaitement au sein d'un système d'aide à la conception et à la réalisation de systèmes. L'étape d'intégration de ce travail, pour l'harmoniser avec RSL n'est pas encore réalisée. Portons un intérêt certain à cette phase ultérieure.

BIBLIOGRAPHIE

BRATKO I., 1986,

"Prolog, programming for artificial intelligence",
International Computer Science Series, n°14224,
Addison-Wesley Publishing Company, 420 pages.

CLOCKSIN W.F., MELLISH C.S., 1981,

"programming in PROLOG",
Springer-Verlag, Berlin, 1981.

DeGROOT D., LINDSTROM G., 1986,

"Logic programming : functions, relations and equations",
Prentice Hall Editions, Englewood Cliffs, New Jersey, 531 pages.

DEMEULEMEESTER R., LALOY J., 1989,

Mémoire présenté en vue de l'obtention du grade de Licencié
et maître en Informatique,
FUNDP, Namur,1989, 6, PP 1-173.

DEVILLE Y., 1987,

"A methodology for logic program construction",
Thèse présentée en vue de l'obtention du grade de Docteur
en Sciences, option Informatique,
FUNDP, Namur,1987.

DUBOIS E., 1984,

"Cadre et méthode de spécifications de sytèmes d'informations fondés sur les types de données",
Thèse de doctorat, Centre de Rech. en Inform., Nancy,
Inst. Nat. Polytech. de Lorraine,1984, 3, 22, pp 1-357.

DUBOIS E., FINANCE J.-P., VAN LAMSWEERDE A., 1985,

"A constructive approach to requirements specification",
Research paper N°1/85, FUNDP Informatique,1985, 2, 4, pp 1-17.

DUBOIS E., VAN LAMSWEERDE A., 1987,

"Making specification processes explicit",
Proc. 4th Int. workshop on softw. spec. and design,
IEEE, Monterey (Calif.),1987, 4, pp 3-4.

EHRIG H., MAHR B., 1985,

"Equations and initial semantics",
in "Fundamentals of algebraic specifications 1",
Ed. Springer-Verlag, EATCS, Monograph on Theoretical Computer Science, 322 pages, 1985.

GUOGEN J.A., THATCHER J.W., WAGNER E.G., 1978,

"An initial Algebra approach to the specification, correctness and implementation of abstract data types",
in "Current trends in programming methodology, vol.4",
Yeh R.T. (Ed.), Prentice-Hall, pp80-149, 1978.

HABRA N., 1986,

"Prototypage en Prolog de spécifications fonctionnelles : une étude de cas",
rapport technique, 1986, F.U.N.D.P. Namur.

HABRA N., 1990,

"A transformationnal approach to functionnal prototyping",
Thèse présentée en vue de l'obtention du grade de Docteur
en Sciences, option Informatique,
FUNDP, Namur, Sept. 1990 (à paraître).

HABRA N., VAN LAMSWEERDE A., 1988,

"Génération de Prototypes Prolog à partir de spécifications
formelles de besoins",
CGLA, "4ième colloque de génie logiciel", 19-21 Octobre 1988,
Paris-France.

HERBRAND J., 1930,

"Researches in the theory of demonstration",
in "From Frege to Gödel : a soruce book in mathematical logic,
1879-1931",
Van Heijenoort J. (Ed.), Harvard University Press, Mass.,
1967, pp 525-581.

LANGELEZ F., PARIS C., 1986,

"Etude et évaluation d'un outil de prototypage",
Mémoire présenté en vue de l'obtention du grade de Licencié
et maître en Informatique, FUNDP, Namur,1986, 6, PP 1-173.

LISKOV B.H., BERZINS V.,

"An appraisal of program specifications",
Comput. Structures Group Memo 141, Lab. for Computer Sciences,
M.I.T. Cambridge, Mass. July 1976.

LISKOV B.H., ZILLES S.P., 1975,

"Specification techniques for data abstractions",

IEEE transactions on software engineering, vol SE-1, 1, pp 7-19.

LLOYD J.W., 1987,

"Foundations of logic programming",

Symbolic computation, Springer-Verlag, préliminaires,

1987, 2nde éd., 1, pp 1-212.

MEYER B., 1984,

"On formalism in specifications",

Technical Report TRC 584-09. University of California (Santa

Barbara), Dept. of Computer Sc., 1984, 6, pp 1-24.

ROBINSON J.A., 1965,

"A machine-oriented logic based on the resolution principle",

J.ACM 12(1), January 1965, pp 23-41.

ROMAN G.-C., 1985,

"A taxonomy of current issues in requirements engeneering",

IEEE, Monterey (Calif.),

1985, 4, pp 14-22.

STERLING L., SHAPIRO E., 1986,

"The art of Prolog",

Advanced Programming Techniques,

The MIT Press, 440 pages.

Facultés Universitaires
Notre-Dame de la Paix
Institut d'informatique
NAMUR

ANNEXES : Liste des spécifications axiomatiques et des procédures Prolog de certains types de données de base

Eric TITECA

Application pratique de la méthode
présentée dans le mémoire

"développement systématique d'une librairie de spécifications algébriques et de son implémentation en Prolog"

Mémoire présenté en vue de
l'obtention du titre de Licencié
et Maître en Informatique

Promoteur: A. VAN LAMSWEERDE
Assistance : N. HABRA

Année académique 1989-1990

ANNEXES : Liste des spécifications axiomatiques et des procédures Prolog de certains types de données de base.

Ce document est divisé en sections modulaires reprenant chacune le développement d'un type particulier, depuis ses spécifications axiomatiques en terme de type de donnée abstrait jusqu'à l'implémentation en langage de programmation logique Prolog.

Afin de faciliter la tache du lecteur, nous avons présenté, pour chaque module, les éléments suivants, dans l'ordre indiqué :

- La spécification syntaxique en TDAs,
- La construction par induction structurelle d'axiomes complets et cohérents (La présentation de cette étape a été gommée dans certains modules, une certaine habitude aidant)

ou,

- La construction par abstraction de ces axiomes, à l'aide d'axiomes de types réutilisés,
- La transformation des équations axiomatiques en procédures Prolog, avec évaluateur,
- La transformation des procédures de l'évaluateur en procédures Prolog logiques, sans évaluation de termes, et
- La liste récapitulative
    * des axiomes TDAs
    * des procédures Prolog avec évaluateur (fonctionnelles),
    * des procédures Prolog sans évaluateur (logiques),
    * des procédures à résultat booléen, simplifiées en Prolog.

Vous trouverez les explications des options choisies lors des différentes étapes dans la troisième partie (application de la théorie) de ce travail.

Voici la liste de chaque module étudié :

A. Les types simples non paramétrés

1. Booléens,
2. Naturels,
3. Entiers,
4. Réels,
5. Caractères.

B. Les types simples paramétrés

1. Séquence (liste, suite)
2. Produit cartésien,
3. Union disjointe.

C. Les types composés paramétrés

1. Ensemble (Set),
2. Pile,
3. Table (tableau, vecteur),
4. Arbre (tree).

MODULE BOOLEEN

TYPE DE DONNEES : BOOLEEN (boolean)

Non paramétré

Inf : Un objet du type BOOLEEN est un élément dont la valeur peut être une valeur de vérité (Vrai ou Faux.

Set : BOOL.

Sigma :

Cons	:	true	:	--› BOOL
		false	:	--› BOOL
Modif	:	not	: BOOL	--› BOOL
		and	: BOOL,BOOL	--› BOOL
		nand	: BOOL,BOOL	--› BOOL
		or	: BOOL,BOOL	--› BOOL
		nor	: BOOL,BOOL	--› BOOL
		xor	: BOOL,BOOL	--› BOOL
		imply	: BOOL,BOOL	--› BOOL
		iff	: BOOL,BOOL	--› BOOL
Sélec	:	before	: BOOL,BOOL	--› BOOL
		after	: BOOL,BOOL	--› BOOL

Explicatif du rôle des fonctions

true : Crée un élément de type Booléen ayant la valeur Vrai
false : Crée un élément de type Booléen ayant la valeur Faux

not : Fournit la valeur de vérité qui est la négation de la valeur argument b
"non b"

and : Fournit la valeur Vrai si les deux arguments booléens valent Vrai, faux sinon.
"b et b'"

nand : Fournit la valeur Vrai si les deux arguments booléens valent Faux, faux sinon.

or : Fournit la valeur Vrai si un des deux arguments booléens vaut Vrai, faux sinon.
"b et/ou b'"

nor : Fournit la valeur Vrai si un des deux arguments booléens vaut Vrai, faux sinon.

xor : Fournit la valeur Vrai si un seul des deux arguments

imply : Fournit la valeur Vrai si le second argument vaut Vrai, sinon fournit la valeur du second argument.
"b => b'"

iff : Fournit la valeur Vrai si les deux arguments ont la même valeur booléenne.
"b <=> b'"

before: Détermine si le premier argument est avant le second dans un ordre déterminé de classement des valeurs booléennes.

after : Détermine le contraire de before

Elaboration des spécifications sémantiques

fonction true() donne Br .

Avec Br : BOOL.

Cette fonction donne une valeur booléènne Br qui est la valeur de vérité Vrai.

Postconditions : Br est la valeur Vrai

Implémentation : true() >prolog> true (constante)

fonction false() donne Br .

Avec Br : BOOL.

Cette fonction donne une valeur booléènne Br qui est la valeur de vérité Faux.

Postconditions : Br est la valeur Faux

Implémentation : false() >prolog> false (constante)

fonction not(B) donne Br .

Avec B,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B est Faux, Faux sinon.

Axiomes_obtenus

```
not(true())  == false()
not(false()) == true()

==(not(true()),false())
==(not(false()),true())

eval(not(true()),Br) <=> Br=false())
eval(not(false()),Br) <=> Br=true())

eval(not(B),Br) <= eval(B,true()) & Br=false())
eval(not(B),Br) <= eval(B,false()) & Br=true())

eval(not(B),false) :- eval(B,true).
eval(not(B),true) :- eval(B,false).

not(true,false).
not(false,true).
```

fonction and(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 et B2 sont tous deux Vrai, faux sinon.

Axiomes_obtenus

```
and(true(),true())   == true()
and(false(),B2)      == false()
and(true(),false())  == false()          (pas de redondance)

==(and(true(),true()),true())
==(and(false(),B2),false())
==(and(true(),false()),false())
```

```
eval(and(true(),true()),Br) <=> Br=true()
eval(and(false(),B2),Br) <=> Br=false()
eval(and(true(),false()),Br) <=> Br=false()

eval(and(B1,B2),Br) <= eval(B1,true()) & eval(B2,true()) &
                       Br=true()
eval(and(B1,B2),Br) <= eval(B1,false()) & Br=false()
eval(and(B1,B2),Br) <= eval(B1,true()) & eval(B2,false()) &
                       Br=false()

eval(and(B1,B2),true) :- eval(B1,true), eval(B2,true).
eval(and(B1,B2),false) :- eval(B1,false).
eval(and(B1,B2),false) :- eval(B1,true), eval(B2,false).

and(true,true,true).
and(false,B2,false).
and(true,false,false).
```

fonction nand(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 vaut Faux ou B2 vaut Faux, faux sinon.

Axiomes_obtenus

```
nand(false(),B2) == true()
nand(B1,false()) == true()
nand(true(),true()) == false()

==(nand(false(),B2),true())
==(nand(B1,false()),true())
==(nand(true(),true()),false())

eval(nand(false(),B2),Br) <=> Br=true()
eval(nand(B1,false()),Br) <=> Br=true()
eval(nand(true(),true()),Br) <=> Br=false()

eval(nand(B1,B2),Br) <= eval(B1,false()) & Br=true()
eval(nand(B1,B2),Br) <= eval(B2,false()) &  Br=true()
eval(nand(B1,B2),Br) <= eval(B1,true()) &  eval(B2,true())
                        & Br=false()

eval(nand(B1,B2),true) :- eval(B1,false).
eval(nand(B1,B2),true) :- eval(B2,false).
eval(nand(B1,B2),false):- eval(B1,true),eval(B2,true).

nand(false,B2,true).
nand(B1,false,true).
nand(true,true,false).
```

```
R.C. nand(B1,B2) == not(and(B1,B2))

     ==(nand(B1,B2),not(and(B1,B2)))
     eval(nand(B1,B2),Br) <=> not(and(B1,B2))

     eval(nand(B1,B2),Br) :- not(and(B1,B2)).

     nand(B1,B2,Br) :- and(B1,B2,B3) not(B3,Br).
```

> fonction or(B1,B2) donne Br .
>
> Avec B1,B2,Br : BOOL,
>
> Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 vaut Vrai ou B2 vaut Vrai, faux sinon.

Axiomes obtenus

```
     or(false(),false()) == false()
     or(true(),B2)       == true()
     or(B1,true())       == true()

     ==(or(false(),false()),false())
     ==(or(true(),B2),true())
     ==(or(B1,true()),true())

     eval(or(false(),false()),Br) <=> Br=false()
     eval(or(true(),B2),Br) <=> Br=true()
     eval(or(B1,true()),Br) <=> Br=true()

     eval(or(B1,B2),Br) <= eval(B1,false()) & eval(B2,false()) &
                           Br=false()
     eval(or(B1,B2),Br) <= eval(B1,true()) & Br=true()
     eval(or(B1,B2),Br) <= eval(B2,true()) & Br=true()

     eval(or(B1,B2),false) :- eval(B1,false), eval(B2,false).
     eval(or(B1,B2),true) :- eval(B1,true).
     eval(or(B1,B2),true) :- eval(B2,true).

     or(false,false,false).
     or(true,B2,true).
     or(B1,true,true).

R.C. or(B1,B2)       == not(and(not(B1),not(B2)))

     ==(or(B1,B2),not(and(not(B1),not(B2))))
     eval(or(B1,B2),Br) <=> eval(not(and(not(B1),not(B2))),Br)
     eval(or(B1,B2),Br) <= eval(not(and(not(B1),not(B2))),Br)
     eval(or(B1,B2),Br) :- eval(not(and(not(B1),not(B2))),Br).

     or(B1,B2,Br) :- not(B1,B3), not(B2,B4), and(B3,B4,B5),
                     not(B5,Br).
```

fonction nor(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 et B2 sont tous deux Faux, faux sinon.

Axiomes obtenus

```
nor(false(),false()) == true()
nor(true(),B2) == false()
nor(B1,true()) == false()

==(nor(false(),false()),true())
==(nor(true(),B2),false())
==(nor(B1,true()),false())

eval(nor(false(),false()),Br) <=> Br=true()
eval(nor(true(),B2),Br) <=> Br=false()
eval(nor(B1,true()),Br) <=> Br=false()

eval(nor(B1,B2),Br) <= eval(B1,false()) & eval(B2,false())
                       & Br=true()
eval(nor(B1,B2),Br) <= eval(B1,true()) & Br=false()
eval(nor(B1,B2),Br) <= eval(B2,true()) & Br=false()

eval(nor(B1,B2),true) :- eval(B1,false), eval(B2,false).
eval(nor(B1,B2),false) :- eval(B1,true).
eval(nor(B1,B2),false) :- eval(B2,true).

nor(false,false,true).
nor(true,B2,false).
nor(B1,true,false).
```

R.C. `nor(B1,B2) == not(or(B1,B2))`

```
==(nor(B1,B2),not(or(B1,B2)))
eval(nor(B1,B2),Br) <=> not(or(B1,B2))

eval(nor(B1,B2),Br) :- not(or(B1,B2)).

nor(B1,B2,Br) :- or(B1,B2,B3) not(B3,Br).
```

fonction xor(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 vaut Vrai et pas B2, ou B2 vaut Vrai et pas B1,
faux sinon.

Axiomes_obtenus

```
if B1=B2  then xor(B1,B2) == false()
if B1<>B2 then xor(B1,B2) == true()

if B1=B2  then ==(xor(B1,B2),false())
if B1<>B2 then ==(xor(B1,B2),true())

if B1=B2  then eval(xor(B1,B2),Br) <=> Br=false()
if B1<>B2 then eval(xor(B1,B2),Br) <=> Br=true()

(eval(xor(B1,B2),Br) <= Br=false()) <= B1=B2
(eval(xor(B1,B2),Br) <= Br=true())  <= B1<>B2

eval(xor(B1,B2),Br) <= B1=B2 & Br=false()
eval(xor(B1,B2),Br) <= B1<>B2 & Br=true()

eval(xor(B1,B1),false).
eval(xor(B1,B2),true).

xor(B1,B1,false).
xor(B1,B2,true).
```

R.C. `xor(B1,B2)  == or(not(or(B1,B2)),and(B1,B2))`

```
==(xor(B1,B2),or(not(or(B1,B2)),and(B1,B2)))
eval(xor(B1,B2),Br) <=>
                eval(or(not(or(B1,B2)),and(B1,B2))),Br)
eval(xor(B1,B2),Br) <=
                eval(or(not(or(B1,B2)),and(B1,B2))),Br)

eval(xor(B1,B2),Br) :-
                eval(or(not(or(B1,B2)),and(B1,B2))),Br).

xor(B1,B2,Br) :- or(B1,B2,B3), not(B3,B4),and(B1,B2,B5),
                 or(B4,B5,Br).
```

fonction imply(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B2 vaut Vrai, sinon prend la valeur de B1.

Axiomes obtenus

```
imply(B1,true())  == true()
imply(B1,false()) == B1

==(imply(B1,true()),true())
==(imply(B1,false()),B1)

eval(imply(B1,true()),Br) <=> Br=true()
eval(imply(B1,false()),Br) <=> eval(B1,Br)

eval(imply(B1,B2),Br) <= eval(B2,true()) & Br=true()
eval(imply(B1,B2),Br) <= eval(B2,false()) & eval(B1,Br)

eval(imply(B1,B2),true) :- eval(B2,true).
eval(imply(B1,B2),Br) :- eval(B2,false), eval(B1,Br).

imply(B1,true,true).
imply(B1,false,B1).
```

R.C. `imply(B1,B2)  == or(not(B1),B2)`

```
==(imply(B1,B2),or(not(B1),B2))
eval(imply(B1,B2),Br) <=> eval(or(not(B1),B2),Br)
eval(imply(B1,B2),Br) <= eval(or(not(B1),B2),Br)

eval(imply(B1,B2),Br) :- eval(or(not(B1),B2),Br).

imply(B1,B2,Br) :- not(B1,B3), or(B3,B2,Br).
```

fonction iff(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 vaut B2.
faux sinon.

Axiomes obtenus

```
if B1=B2 then  iff(B1,B2) == true()
if B1<>B2 then iff(B1,B2) == false()

if B1=B2 then  ==(iff(B1,B2),true())
if B1<>B2 then ==(iff(B1,B2),false())

if B1=B2 then  eval(iff(B1,B2),Br) <=> Br=true()
if B1<>B2 then eval(iff(B1,B2),Br) <=> Br=false()

(eval(iff(B1,B2),Br) <= Br=true()) <= B1=B2
(eval(iff(B1,B2),Br) <= Br=false()) <= B1<>B2

eval(iff(B1,B2),Br) <= B1=B2 & Br=true()
eval(iff(B1,B2),Br) <= B1<>B2 & Br=false()

eval(iff(B1,B1),true).
eval(iff(B1,B2),false).

iff(B1,B1,true).
iff(B1,B2,false).
```

R.C.
```
iff(B1,B2)  == and(imply(B1,B2),imply(B2,B1))

==(iff(B1,B2),and(imply(B1,B2),imply(B2,B1)))
eval(iff(B1,B2),Br) <=>
                eval(and(imply(B1,B2),imply(B2,B1)),Br)

eval(iff(B1,B2),Br) :-
                eval(and(imply(B1,B2),imply(B2,B1)),Br).

iff(B1,B2,Br) :- imply(B1,B2,B3),imply(B2,B1,B4),
                 and(B3,B4,Br).
```

R.C.
```
iff(B1,B2)  == not(xor(B1,B2)

==(iff(B1,B2),not(xor(B1,B2)))
eval(iff(B1,B2),Br) <=> eval(not(xor(B1,B2)),Br)

eval(iff(B1,B2),Br) :- eval(not(xor(B1,B2)),Br).

iff(B1,B2,Br) :- xor(B1,B2,B3), not(B3,Br).
```

fonction before(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 est Faux, c'est-à-dire si B1 est après B2 (Faux étant avant Vrai selon un ordre fixé par principe).

Axiomes_obtenus

```
before(false(),B2) == true()
before(true(),B2)  == false()

==(before(false(),B2),true())
==(before(true(),B2),false())

eval(before(false(),B2),Br) <=> Br=true()
eval(before(true(),B2),Br) <=> Br=false()

eval(before(B1,B2),Br) <= eval(B1,false()) & Br=true()
eval(before(B1,B2),Br) <= eval(B1,true()) & Br=false()

eval(before(B1,B2),true) :- eval(B1,false).
eval(before(B1,B2),false) :- eval(B1,true).

before(false,_,true).
before(true,_,false).
```

R.C. before(B1,B2) == not(B1)

```
==(before(B1,B2),not(B1))
eval(before(B1,B2),Br) <=> eval(not(B1),Br)

eval(before(B1,B2),Br) :- eval(not(B1),Br).

before(B1,_,Br) :- not(B1,Br).
```

fonction after(B1,B2) donne Br .

Avec B1,B2,Br : BOOL,

Cette fonction donne un élément de type booléen Br qui prend la valeur Vrai si B1 est Vrai, c'est-à-dire si B1 est avant B2 (Faux étant avant Vrai selon un ordre fixé par principe).

Axiomes_obtenus

```
after(false(),B2)   == false()
after(true(),B2)    == true()

==(after(false(),B2),false())
==(after(true(),B2),true())

eval(after(false(),B2),Br) <=> Br=false()
eval(after(true(),B2),Br) <=> Br=true()

eval(after(B1,B2),Br) <= eval(B1,false()) & Br=false()
eval(after(B1,B2),Br) <= eval(B1,true()) & Br=true()

eval(after(B1,B2),false) :- eval(B1,false).
eval(after(B1,B2),true) :- eval(B1,true).

after(false,_,false).
after(true,_,true).
```

R.C. `after(B1,B2)  == B1`

```
==(after(B1,B2),B1)
eval(after(B1,B2),Br) <=> eval(B1,Br)

eval(after(B1,B2),Br) :- eval(B1,Br).

after(B1,_,B1).
```

R.C. `after(B1,B2)  == not(before(B1,B2))`

```
==(after(B1,B2),not(before(B1,B2)))
eval(after(B1,B2),Br) <=> eval(not(before(B1,B2)),Br)

eval(after(B1,B2),Br) :- eval(not(before(B1,B2)),Br).

after(B1,B2,Br) :- before(B1,B2,B3), not(B3,Br).
```

Récapitulatif des axiomes ADTs obtenus

```
not(true())  == false()
not(false()) == true()

and(true(),true())   == true()
and(false(),B2)      == false()
and(true(),false())  == false()

nand(false(),B2) == true()
nand(B1,false()) == true()
nand(true(),true()) == false()

or(false(),false()) == false()
or(true(),B2)       == true()
or(B1,true())       == true()

nor(false(),false()) == true()
nor(true(),B2) == false()
nor(B1,true()) == false()

if B1=B2  then xor(B1,B2) == false()
if B1<>B2 then xor(B1,B2) == true()

if B1=B2  then imply(B1,true())  == true()
if B1<>B2 then imply(B1,false()) == B1

iff(B1,B1)         == true()
iff(B1,B2)         == false()

before(false(),B2) == true()
before(true(),B2)  == false()

after(B1,B2)  == B1
```

Récapitulatif des procédures PROLOG avec évaluateur

```
not(B)

     eval(not(B),false) :- eval(B,true).
     eval(not(B),true) :- eval(B,false).

and(B1,B2)

     eval(and(B1,B2),true) :- eval(B1,true), eval(B2,true).
     eval(and(B1,B2),false) :- eval(B1,false).
     eval(and(B1,B2),false) :- eval(B1,true), eval(B2,false).

nand(B1,B2)

     eval(nand(B1,B2),true) :- eval(B1,false).
     eval(nand(B1,B2),true) :- eval(B2,false).
     eval(nand(B1,B2),false):- eval(B1,true),eval(B2,true).

or(B1,B2)

     eval(or(B1,B2),false) :- eval(B1,false), eval(B2,false).
     eval(or(B1,B2),true) :- eval(B1,true).
     eval(or(B1,B2),true) :- eval(B2,true).

nor(B1,B2)

     eval(nor(B1,B2),true) :- eval(B1,false), eval(B2,false).
     eval(nor(B1,B2),false) :- eval(B1,true).
     eval(nor(B1,B2),false) :- eval(B2,true).

xor(B1,B2)

     eval(xor(B1,B1),false).
     eval(xor(B1,B2),true).

imply(B1,B2)

     eval(imply(B1,B2),true) :- eval(B2,true).
     eval(imply(B1,B2),Br) :- eval(B2,false), eval(B1,Br).

iff(B1,B2)

     eval(iff(B1,B1),true).
     eval(iff(B1,B2),false).

before(B1,B2)

     eval(before(B1,B2),true) :- eval(B1,false).
     eval(before(B1,B2),false) :- eval(B1,true).

after(B1,B2)

     eval(after(B1,B2),false) :- eval(B1,false).
     eval(after(B1,B2),true) :- eval(B1,true).
```

Récapitulatif des procédures PROLOG sans évaluateur

```
not(true,false).
not(false,true).

and(true,true,true).
and(false,B2,false).
and(true,false,false).

nand(false,B2,true).
nand(B1,false,true).
nand(true,true,false).

or(false,false,false).
or(true,B2,true).
or(B1,true,true).

nor(false,false,true).
nor(true,B2,false).
nor(B1,true,false).

xor(B1,B1,false).
xor(B1,B2,true).

imply(B1,true,true).
imply(B1,false,B1).

iff(B1,B1,true).
iff(B1,B2,false).

before(false,_,true).
before(true,_,false).

after(false,_,false).
after(true,_,true).
```

MODULE NATUREL

TYPE DE DONNEES : NATUREL (natural)

Non paramétré

Inf : Un objet du type NATUREL est un élément dont la valeur appartient à l'ensemble mathématique N, c'est-à-dire compris entre zero et l'infini, par pas de 1 unité.

Set : NAT, BOOL.

Sigma :

```
    Cons   :  zero   :            --> NAT
              succ   :  NAT       --> NAT

    Modif  :  pred   :  NAT       --> NAT
              add    :  NAT,NAT   --> NAT
              soust  :  NAT,NAT   --> NAT
              mult   :  NAT,NAT   --> NAT
              div    :  NAT,NAT   --> INT
              mod    :  NAT,NAT   --> INT

    Sélec  :  is_less  :  NAT,NAT --> BOOL
              is_great :  NAT,NAT --> BOOL
              is_eq    :  NAT,NAT --> BOOL
              is_leq   :  NAT,NAT --> BOOL
              is_geq   :  NAT,NAT --> BOOL
              is_diff  :  NAT,NAT --> BOOL
              is_pair  :  NAT     --> BOOL
              is_impair:  NAT     --> BOOL
```

Explicatif du rôle des fonctions

zero	: Fournit la valeur entière nulle
succ	: Fournit la valeur entière positive suivante d'une autre
pred	: Fournit la valeur entière positive précédente d'une autre
add	: Fournit la valeur entière positive résultant de l'addition de deux autres
soust	: Fournit la valeur entière positive résultant de la soustraction de deux autres
mult	: Fournit la valeur entière positive résultant de la multiplication de deux autres
div	: Fournit la valeur entière positive résultant de la division de deux autres
mod	: Fournit la valeur entière positive reste de la division de deux autres
is_less	: Détermine si une valeur entière positive est plus petite qu'une autre
is_great	: Détermine si une valeur entière positive est plus grande qu'une autre
is_eq	: Détermine si une valeur entière positive est égale à une autre
is_leq	: Détermine si une valeur entière positive est plus petite ou égale à une autre
is_geq	: Détermine si une valeur entière positive est plus grande ou égale à une autre
is_diff	: Détermine si une valeur entière positive est différente d'une autre
is_pair	: Détermine si une valeur entière positive est paire (multiple de 2, succ(succ(zero)))
is_impair	: Détermine si une valeur entière positive est impaire (pas multiple de 2, succ(succ(zero)))

fonction zero() donne Nr .

Avec Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui prend la valeur minimum des naturels, zero (Nr=0).

Implémentation : zero() ›prolog› 0

```
eval(zero(),0).
```

fonction succ(N) donne Nr .

Avec N,Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui prend la valeur suivante de N dans l'ensemble mathématique N (Nr=N+1).

Implémentation : succ(N) ›prolog› N + 1

```
eval(succ(N),Nr) :- eval(N,N1), Nr is N1+1.

succ(N,Nr) :- Nr is N+1.
```

fonction pred(N) donne Nr .

Avec N,Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui prend la valeur précédente de N dans l'ensemble mathématique N (Nr=N-1).

Axiomes obtenus

```
pred(zero())      == zero()
pred(succ(N1))    == N1

==(pred(zero()),zero())
==(pred(succ(N1)),N1)

eval(pred(zero()),Nr) ‹=› Nr=zero()
eval(pred(succ(N1)),Nr) ‹=› eval(N1,Nr)
```

```
eval(pred(N),Nr) <= eval(N,zero()) & Nr=zero()
eval(pred(N),Nr) <= eval(N,succ(N1)) & eval(N1,Nr)

eval(pred(N),0) :- eval(N,0).
eval(pred(N),Nr) :- eval(N,N2), Nr is N2-1.

pred(0,0).
pred(N,Nr) :-  Nr is N-1.
```

fonction add(N,N2) donne Nr.

Avec N,N2,Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui est le résultat de l'addition de N et N2.

Axiomes obtenus

```
add(N,zero())    == N
add(N,succ(N1))  == succ(add(N,N1))

==(add(N,zero()),N)
==(add(N,succ(N1)),succ(add(N,N1)))

eval(add(N,zero()),Nr) <=> eval(N,Nr)
eval(add(N,succ(N1)),Nr) <=> eval(succ(add(N,N1)),Nr)

eval(add(N,N2),Nr) <= eval(N2,zero()) & eval(N,Nr)
eval(add(N,N2),Nr) <= eval(N2,succ(N1)) &
                      eval(succ(add(N,N1)),Nr)

eval(add(N,N2),Nr) :- eval(N2,0),eval(N,Nr).
eval(add(N,N2),Nr) :- eval(N2,N3), N1 is N3-1,
                      eval(succ(add(N,N1)),Nr).

eval(add(N,N2),Nr) :- eval(N2,0),eval(N,Nr).
eval(add(N,N2),Nr) :- eval(N2,N3), N1 is N3-1,
                      eval(add(N,N1),N3), Nr is N3+1.

add(N,0,N).
add(N,N2,Nr) :- N1 is N2-1, add(N,N1,N3), Nr is N3+1.
```

fonction soust(N,N2) donne Nr.

Avec N,N2,Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui est le résultat de la soustraction de N2 à N.

Axiomes_obtenus

```
soust(N,zero())      == N
soust(N,succ(N1))    == pred(soust(N,N1))

==(soust(N,zero()),N)
==(soust(N,succ(N1)),pred(soust(N,N1)))

eval(soust(N,zero()),Nr) <=> eval(N,Nr)
eval(soust(N,succ(N1)),Nr) <=> eval(pred(soust(N,N1)),Nr)

eval(soust(N,N2),Nr) <= eval(N2,zero()) & eval(N,Nr)
eval(soust(N,N2),Nr) <= eval(N2,succ(N1)) &
                        eval(pred(soust(N,N1)),Nr)

eval(soust(N,N2),Nr) :- eval(N2,0), eval(N,Nr).
eval(soust(N,N2),Nr) :- eval(N2,N3), N1 is N3-1,
                        eval(pred(soust(N,N1)),Nr).

soust(N,0,N).
soust(N,N2,Nr) :- N1 is N2-1, soust(N,N1,N3), pred(N3,Nr).
```

fonction mult(N,N2) donne Nr.

Avec N,N2,Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui est le résultat de la multiplication de N et N2.

Axiomes_obtenus

```
mult(N,zero())    == zero()
mult(N,succ(N1)) == add(N,mult(N,N1))

==(mult(N,zero()),zero())
==(mult(N,succ(N1)),add(N,mult(N,N1)))

eval(mult(N,zero()),Nr) <=> Nr=zero()
eval(mult(N,succ(N1)),Nr) <=> eval(add(N,mult(N,N1)),Nr)
```

```
eval(mult(N,N2),Nr) <= eval(N2,zero()) & Nr=zero()
eval(mult(N,N2),Nr) <= eval(N2,succ(N1)) &
                       eval(add(N,mult(N,N1)),Nr)

eval(mult(N,N2),0) :- eval(N2,0).
eval(mult(N,N2),Nr) :- eval(N2,N3), N1 is N3-1,
                       eval(add(N,mult(N,N1)),Nr).

mult(N,0,0).
mult(N,N2,Nr) :- N1 is N2-1, mult(N,N1,N3), add(N,N3,Nr).
```

on peut y rajouter, en tête, pour l'efficacité, l'axiome :

```
mult(0,N,0).
```

fonction div(N,N2) donne Nr.

Avec N,N2,Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui est le résultat naturel de la division de N par N2.

Axiomes_obtenus

```
div(N,zero())  == ERREUR "Division par zero illégale"
div(zero(),N2) == zero()
if eval(is_less(succ(N1),N2),true()) then
        div(succ(N1),N2)== zero()
if eval(is_less(succ(N1),N2),false()) then
        div(succ(N1),N2)== succ(div(soust(succ(N1),N2),N2)))

div(zero(),N2) == zero()
if eval(is_less(N,N2),true()) then
          div(N,N2)== zero()
if eval(is_less(N,N2),false()) then
          div(N,N2)== succ(div(soust(N,N2),N2))

==(div(zero(),N2),zero())
if eval(is_less(N,N2),true()) then
          ==(div(N,N2),zero())
if eval(is_less(N,N2),false()) then
          ==(div(N,N2),succ(div(soust(N,N2),N2)))

eval(div(zero(),N2),Nr) <=> Nr=zero()
if eval(is_less(N,N2),true()) then
   eval(div(N,N2),Nr) <=> Nr=zero()
if eval(is_less(N,N2),false()) then
   eval(div(N,N2),Nr) <=> eval(succ(div(soust(N,N2),N2)),Nr)
```

```
eval(div(N,N2),Nr) <= eval(N,zero()) & Nr=zero()
if eval(is_less(N,N2),true()) then
   eval(div(N,N2),Nr) <= Nr=zero()
if eval(is_less(N,N2),false()) then
   eval(div(N,N2),Nr) <= eval(succ(div(soust(N,N2),N2)),Nr)

eval(div(N,N2),Nr) <= eval(N,zero()) & Nr=zero()
(eval(div(N,N2),Nr) <= Nr=zero())
                <= eval(is_less(N,N2),true())
(eval(div(N,N2),Nr) <= eval(succ(div(soust(N,N2),N2)),Nr))
                <= eval(is_less(N,N2),false())

eval(div(N,N2),Nr) <= eval(N,zero()) & Nr=zero()
eval(div(N,N2),Nr) <= eval(is_less(N,N2),true())
                       & Nr=zero())
eval(div(N,N2),Nr) <= eval(is_less(N,N2),false())
                & eval(succ(div(soust(N,N2),N2)),Nr)

eval(div(N,N2),Nr) <= eval(N,zero()) & Nr=zero()
eval(div(N,N2),Nr) <= eval(is_less(N,N2),true())
                       & Nr=zero())
eval(div(N,N2),Nr) <= eval(is_less(N,N2),false())
                & eval(succ(div(soust(N,N2),N2)),Nr)

eval(div(N,N2),0) :- eval(N,0).
eval(div(N,N2),0) :- eval(is_less(N,N2),true).
eval(div(N,N2),Nr) :- eval(is_less(N,N2),false),
                      eval(succ(div(soust(N,N2),N2)),Nr).

eval(div(N,N2),0) :- eval(N,0).
eval(div(N,N2),0) :- eval(is_less(N,N2),true).
eval(div(N,N2),Nr) :- eval(is_less(N,N2),false),
                  eval(div(soust(N,N2),N2),N3), Nr is N3+1.

div(0,N2,0).
div(N,N2,0) :- is_less(N,N2,true).
div(N,N2,Nr) :- is_less(N,N2,false),soust(N,N2,N4),
                div(N4,N2,N3), Nr is N3+1.

div(0,N2,0).
div(N,N2,0) :- is_less(N,N2).
div(N,N2,Nr) :- not(is_less(N,N2)),soust(N,N2,N4),
                div(N4,N2,N3), Nr is N3+1.
```

fonction mod(N,N2) donne Nr.

Avec N,N2,Nr : NAT.

Cette fonction donne un élément de type naturel Nr qui est le reste de la division de N par N2.

Axiomes_obtenus

```
mod(N,zero())  == ERREUR "Division par zero illégale"
mod(zero(),N2) == zero()
if eval(is_less(succ(N1),N2),true()) then
        mod(succ(N1),N2) == succ(N1)
if eval(is_less(succ(N1),N2),false()) then
        mod(succ(N1),N2) == mod(soust(succ(N1),N2),N2)

mod(zero(),N2) == zero()
if eval(is_less(N,N2),true()) then
          mod(N,N2) == N
if eval(is_less(N,N2),false()) then
          mod(N,N2) == mod(soust(N,N2),N2)

==(mod(zero(),N2),zero())
if eval(is_less(N,N2),true()) then
          ==(mod(N,N2),N)
if eval(is_less(N,N2),false()) then
          ==(mod(N,N2),mod(soust(N,N2),N2))

eval(mod(zero(),N2),Nr) <=> Nr=zero()
if eval(is_less(N,N2),true()) then
         eval(mod(N,N2),Nr) <=> eval(N,Nr)
if eval(is_less(N,N2),false()) then
         eval(mod(N,N2),Nr) <=> eval(mod(soust(N,N2),N2),Nr)

eval(mod(N,N2),Nr) <= eval(N,zero()) & Nr=zero()
if eval(is_less(N,N2),true()) then
         eval(mod(N,N2),Nr) <= eval(N,Nr)
if eval(is_less(N,N2),false()) then
         eval(mod(N,N2),Nr) <= eval(mod(soust(N,N2),N2),Nr)

eval(mod(N,N2),Nr) <= eval(N,zero()) & Nr=zero()
(eval(mod(N,N2),Nr) <= eval(N,Nr)) <=
                  eval(is_less(N,N2),true())
(eval(mod(N,N2),Nr) <= eval(mod(soust(N,N2),N2),Nr)) <=
                  eval(is_less(N,N2),false())

eval(mod(N,N2),Nr) <= eval(N,zero()) & Nr=zero()
eval(mod(N,N2),Nr) <= eval(is_less(N,N2),true())
                      & eval(N,Nr)
eval(mod(N,N2),Nr) <= eval(is_less(N,N2),false()) &
                      eval(mod(soust(N,N2),N2),Nr)
```

```
eval(mod(N,N2),0) :- eval(N,0).
eval(mod(N,N2),Nr) :- eval(is_less(N,N2),true),
                      eval(N,Nr).
eval(mod(N,N2),Nr) :- eval(is_less(N,N2),false),
                      eval(mod(soust(N,N2),N2),Nr).

mod(0,N2,0).
mod(N,N2,N) :- is_less(N,N2,true).
mod(N,N2,Nr) :- is_less(N,N2,false),soust(N,N2,N4),
                mod(N4,N2,Nr).

mod(0,N2,0).
mod(N,N2,N) :- is_less(N,N2).
mod(N,N2,Nr) :- not(is_less(N,N2)),soust(N,N2,N4),
                mod(N4,N2,Nr).
```

fonction is_equal(N,N2) donne Br.

Avec N,N2 : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est égal à N2, faux sinon.

Axiomes_obtenus

```
is_equal(zero(),zero()) == true()
is_equal(zero(),succ(N3)) == false()
is_equal(succ(N1),zero()) == false()
is_equal(succ(N1),succ(N3)) == is_equal(N1,N3)

==(is_equal(zero(),zero()),true())
==(is_equal(zero(),succ(N3)),false())
==(is_equal(succ(N1),zero()),false())
==(is_equal(succ(N1),succ(N3)),is_equal(N1,N3))

eval(is_equal(zero(),zero()),Br) <=> Br=true()
eval(is_equal(zero(),succ(N3)),Br) <=> Br=false()
eval(is_equal(succ(N1),zero()),Br) <=> Br=false()
eval(is_equal(succ(N1),succ(N3)),Br) <=>
                          eval(is_equal(N1,N3),Br)

eval(is_equal(N,N2),Br) <= eval(N,zero()) & eval(N2,zero())
                           & Br=true()
eval(is_equal(N,N2),Br) <= eval(N,zero()) &
                           eval(N2,succ(N3)) & Br=false()
eval(is_equal(N,N2),Br) <= eval(N,succ(N1)) &
                           eval(N2,zero()) & Br=false()
eval(is_equal(N,N2),Br) <= eval(N,succ(N1)) &
                eval(N2,succ(N3)) & eval(is_equal(N1,N3),Br)
```

```
eval(is_equal(N,N2),true) :- eval(N,0), eval(N2,0).
eval(is_equal(N,N2),false) :- eval(N,0).
eval(is_equal(N,N2),false) :- eval(N2,0).
eval(is_equal(N,N2),Br) :- eval(N,N4), N1 is N4-1,
          eval(N2,N5), N3 is N5-1, eval(is_equal(N1,N3),Br).

is_equal(0,0,true).
is_equal(0,N2,false).
is_equal(N,0,false).
is_equal(N,N2,Br) :- N1 is N-1, N3 is N2-1,
                     is_equal(N1,N3,Br).
```

R.C. `is_equal(N,N2) == and(is_leq(N,N2),is_leq(N2,N))`

(à condition que is_leq ne soit pas défini en terme de is_equal !)

fonction is_less(N,N2) donne Br.

Avec N,N2 : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est plus petit (plus proche de zero) que N2, faux sinon.

Axiomes_obtenus

```
is_less(zero(),N2) == true()
is_less(succ(N1),zero()) == false()
is_less(succ(N1),succ(N3)) == is_less(N1,N3)

==(is_less(zero(),N2),true())
==(is_less(succ(N1),zero()),false())
==(is_less(succ(N1),succ(N3)),is_less(N1,N3))

eval(is_less(zero(),N2),R) <=> R=true()
eval(is_less(succ(N1),zero()),R) <=> R=false()
eval(is_less(succ(N1),succ(N3)),R) <=> eval(is_less(N1,N3),R)

eval(is_less(N,N2),R) <= eval(N,zero()) & eval(N2,N3) &
                         R=true()
eval(is_less(N,N2),R) <= eval(N,succ(N1)) & eval(N2,zero())
                         & R=false()
eval(is_less(N,N2),R) <= eval(N,succ(N1)) & eval(N2,succ(N3)) &
                         eval(is_less(N1,N3),R)

eval(is_less(N,N2),true) :- eval(N,0), eval(N2,N3).
eval(is_less(N,N2),false) :- eval(N,N3),N1 is N3-1, eval(N2,0).
eval(is_less(N,N2),R) :- eval(N,N4), N1 is N4 - 1, eval(N2,N5),
                         N3 is N5 - 1, eval(is_less(N1,N3),R).
```

```
eval(is_less(N,N2),true) :- eval(N,0).
eval(is_less(N,N2),false) :- eval(N2,0).
eval(is_less(N,N2),R) :- eval(N,N4), N1 is N4 - 1, eval(N2,N5),
                         N3 is N5 - 1, eval(is_less(N1,N3),R).

is_less(0,N2,true).
is_less(N,0,false).
is_less(N,N2,Br) :- N1 is N-1, N3 is N2-1,is_less(N1,N3,Br).
```

fonction is_leq(N,N2) donne Br.

Avec N,N2 : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est plus petit (plus proche de zero) ou égal à N2, faux sinon.

Axiomes obtenus

```
is_leq(zero(),N2)          == true()
is_leq(N,zero())           == is_equal(N,zero())
is_leq(succ(N1),succ(N3)) == is_leq(N1,N3)

==(is_leq(zero(),N2),true())
==(is_leq(N,zero()),is_equal(N,zero()))
==(is_leq(succ(N1),succ(N3)),is_leq(N1,N3))

eval(is_leq(zero(),N2),R) <=> R=true()
eval(is_leq(N,zero()),R)  <=> eval(is_equal(N,zero()),R)
eval(is_leq(succ(N1),succ(N3)),R) <=> eval(is_leq(N1,N3),R)

eval(is_leq(N,N2),R) <= eval(N,zero()) & eval(N2,N3) &
                        R=true()
eval(is_leq(N,N2),R) <= eval(N2,zero()) &
                        eval(is_equal(N,zero()),R)
eval(is_leq(N,N2),R) <= eval(N,succ(N1)) & eval(N2,succ(N3)) &
                        eval(is_leq(N1,N3),R)

eval(is_leq(N,N2),true) :- eval(N,0), eval(N2,N3).
eval(is_leq(N,N2),R) :- eval(N2,0),eval(is_equal(N,0),R).
eval(is_leq(N,N2),R) :- eval(N,N4), N1 is N4 - 1, eval(N2,N5),
                        N3 is N5 - 1, eval(is_leq(N1,N3),R).

eval(is_leq(N,N2),true) :- eval(N,0).
eval(is_leq(N,N2),R) :- eval(N2,0),eval(is_equal(N,0),R).
eval(is_leq(N,N2),R) :- eval(N,N4), N1 is N4 - 1, eval(N2,N5),
                        N3 is N5 - 1, eval(is_leq(N1,N3),R).

is_leq(0,N2,true).
is_leq(N,0,R) :- is_equal(N,0,R).
is_leq(N,N2,Br) :- N1 is N-1, N3 is N2-1,is_leq(N1,N3,Br).
```

fonction is_great(N,N2) donne Br.

Avec N,N2 : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est plus grand (plus éloigné de zero) que N2, faux sinon.

Axiomes_obtenus

```
is_great(zero(),succ(N3))    == false()
is_great(succ(N1),zero())    == true()
is_great(zero(),zero())      == false()
is_great(succ(N1),succ(N3)) == is_great(N1,N3)

is_great(zero(),N2)          == false()
is_great(N,zero())           == true()
is_great(succ(N1),succ(N3)) == is_great(N1,N3)

==(is_great(zero(),N2),false())
==(is_great(N,zero()),true())
==(is_great(succ(N1),succ(N3)),is_great(N1,N3))

eval(is_great(zero(),N2),R) <=> R=false()
eval(is_great(N,zero()),R) <=> R=true()
eval(is_great(succ(N1),succ(N3)),R) <=> eval(is_great(N1,N3),R)

eval(is_great(N,N2),R) <= eval(N,zero()) & R=false()
eval(is_great(N,N2),R) <= eval(N2,zero()) & R=true()
eval(is_great(N,N2),R) <= eval(N,succ(N1)) & eval(N2,succ(N3))
                          & eval(is_great(N1,N3),R)

eval(is_great(N,N2),false) :- eval(N,0).
eval(is_great(N,N2),true) :- eval(N2,0).
eval(is_great(N,N2),R) :- eval(N,N4), N1 is N4-1, eval(N2,N5),
                          N3 is N5-1, eval(is_great(N1,N3),R).

eval(is_great(N,N2),false) :- eval(N,0).
eval(is_great(N,N2),true) :- eval(N2,0).
eval(is_great(N,N2),R) :- eval(N,N4), N1 is N4-1, eval(N2,N5),
                          N3 is N5-1, eval(is_great(N1,N3),R).

is_great(0,N2,false).
is_great(N,0,true).
is_great(N,N2,R) :- N1 is N-1, N3 is N2-1, is_great(N1,N3,R).
```

fonction is_geq(N,N2) donne Br.

Avec N,N2 : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est plus grand (plus éloigné de zero) ou égal à N2, faux sinon.

Axiomes_obtenus

```
is_geq(zero(),N2)          == is_equal(zero(),N2)
is_geq(N,zero())           == true()
is_geq(zero(),zero())      == true()
is_geq(succ(N1),succ(N3)) == is_geq(N1,N3)

is_geq(zero(),N2)          == is_equal(zero(),N2)
is_geq(N,zero())           == true()
is_geq(succ(N1),succ(N3)) == is_geq(N1,N3)

==(is_geq(zero(),N2),is_equal(zero(),N2))
==(is_geq(N,zero()),true())
==(is_geq(succ(N1),succ(N3)),is_geq(N1,N3))

eval(is_geq(zero(),N2),R) <=> eval(is_equal(zero(),N2),R)
eval(is_geq(N,zero()),R) <=> R=true()
eval(is_geq(succ(N1),succ(N3)),R) <=> eval(is_geq(N1,N3),R)

eval(is_geq(N,N2),R) <= eval(N,zero()) &
                        eval(is_equal(zero(),N2),R)
eval(is_geq(N,N2),R) <= eval(N2,zero()) & R=true()
eval(is_geq(N,N2),R) <= eval(N,succ(N1)) & eval(N2,succ(N3))
                        & eval(is_geq(N1,N3),R)

eval(is_geq(N,N2),R) :- eval(N,0), eval(is_equal(0,N2),R).
eval(is_geq(N,N2),true) :- eval(N2,0).
eval(is_geq(N,N2),R) :- eval(N,N4), N1 is N4-1, eval(N2,N5),
                          N3 is N5-1, eval(is_geq(N1,N3),R).

eval(is_geq(N,N2),R) :- eval(N,0), eval(is_equal(0,N2),R).
eval(is_geq(N,N2),true) :- eval(N2,0).
eval(is_geq(N,N2),R) :- eval(N,N4), N1 is N4-1, eval(N2,N5),
                          N3 is N5-1, eval(is_geq(N1,N3),R).

is_geq(0,N2,R) :- is_equal(0,N2,R).
is_geq(N,0,true).
is_geq(N,N2,R) :- N1 is N-1, N3 is N2-1, is_geq(N1,N3,R).
```

fonction is_diff(N,N2) donne Br.

Avec N,N2 : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est différent de N2, faux sinon.

Axiomes_obtenus

```
is_diff(zero(),zero()) == false()
is_diff(zero(),succ(N3)) == true()
is_diff(succ(N1),zero()) == true()
is_diff(succ(N1),succ(N3)) == is_equal(N1,N3)

==(is_diff(zero(),zero()),false())
==(is_diff(zero(),succ(N3)),true())
==(is_diff(succ(N1),zero()),true())
==(is_diff(succ(N1),succ(N3)),is_equal(N1,N3))

eval(is_diff(zero(),zero()),Br) <=> Br=false()
eval(is_diff(zero(),succ(N3)),Br) <=> Br=true()
eval(is_diff(succ(N1),zero()),Br) <=> Br=true()
eval(is_diff(succ(N1),succ(N3)),Br) <=>
                          eval(is_equal(N1,N3),Br)

eval(is_diff(N,N2),Br) <= eval(N,zero()) & eval(N2,zero())
                          & Br=false()
eval(is_diff(N,N2),Br) <= eval(N,zero()) &
                          eval(N2,succ(N3)) & Br=true()
eval(is_diff(N,N2),Br) <= eval(N,succ(N1)) &
                          eval(N2,zero()) & Br=true()
eval(is_diff(N,N2),Br) <= eval(N,succ(N1)) &
               eval(N2,succ(N3)) & eval(is_equal(N1,N3),Br)
```

fonction is_diff(N,N2) donne Br.

Avec N,N2 : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est différent de N2, faux sinon.

Axiomes_obtenus

```
is_diff(zero(),zero()) == false()
is_diff(zero(),succ(N3)) == true()
is_diff(succ(N1),zero()) == true()
is_diff(succ(N1),succ(N3)) == is_equal(N1,N3)

==(is_diff(zero(),zero()),false())
==(is_diff(zero(),succ(N3)),true())
==(is_diff(succ(N1),zero()),true())
==(is_diff(succ(N1),succ(N3)),is_equal(N1,N3))

eval(is_diff(zero(),zero()),Br) <=> Br=false()
eval(is_diff(zero(),succ(N3)),Br) <=> Br=true()
eval(is_diff(succ(N1),zero()),Br) <=> Br=true()
eval(is_diff(succ(N1),succ(N3)),Br) <=>
                          eval(is_equal(N1,N3),Br)

eval(is_diff(N,N2),Br) <= eval(N,zero()) & eval(N2,zero())
                           & Br=false()
eval(is_diff(N,N2),Br) <= eval(N,zero()) &
                           eval(N2,succ(N3)) & Br=true()
eval(is_diff(N,N2),Br) <= eval(N,succ(N1)) &
                           eval(N2,zero()) & Br=true()
eval(is_diff(N,N2),Br) <= eval(N,succ(N1)) &
               eval(N2,succ(N3)) & eval(is_equal(N1,N3),Br)
```

```
eval(is_diff(N,N2),false) :- eval(N,0), eval(N2,0).
eval(is_diff(N,N2),true) :- eval(N,0).
eval(is_diff(N,N2),true) :- eval(N2,0).
eval(is_diff(N,N2),Br) :- eval(N,N4), N1 is N4-1,
          eval(N2,N5), N3 is N5-1, eval(is_equal(N1,N3),Br).

is_diff(0,0,false).
is_diff(0,N2,true).
is_diff(N,0,true).
is_diff(N,N2,Br) :- N1 is N-1, N3 is N2-1,
                    is_equal(N1,N3,Br).
```

R.C. `is_diff(N,N2) == not(is_equal(N,N2))`

```
is_diff(N,N2,R) :- is_equal(N,N2,R2), not(R2,R).
```

fonction is_pair(N) donne Br.

Avec N : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est pair (reste de la division de N par 2 égal à 0), faux sinon.

Axiomes_obtenus

```
is_pair(zero()) == true()
is_pair(succ(zero()) == false()
is_pair(succ(N1)) == not(is_pair(N1))

==(is_pair(zero()),true())
==(is_pair(succ(zero()),false())
==(is_pair(succ(N1)),not(is_pair(N1)))

eval(is_pair(zero()),R) <=> R=true()
eval(is_pair(succ(zero()),R) <=> R=false()
eval(is_pair(succ(N1)),R) <=> eval(not(is_pair(N1)),R)

eval(is_pair(N,R) <= eval(N,zero()) & R=true()
eval(is_pair(N,R) <= eval(N,succ(zero()) & R=false()
eval(is_pair(N,R) <= eval(N,succ(N1)) & eval(not(is_pair(N1)),R)

eval(is_pair(N,true) :- eval(N,0).
eval(is_pair(N,false) :- eval(N,N1), N3 is N1-1, N3=0.
eval(is_pair(N,R) :- eval(N,N3), N1 is N3-1,
                     eval(not(is_pair(N1)),R).

is_pair(0,true).
is_pair(1,false).
is_pair(N,R) :- N1 is N-1, is_pair(N1,R2), not(R2,R).
```

```
R.C. is_pair(N) == is_equal(zero(),mod(N,succ(succ(zero()))))
     ==(is_pair(N),is_equal(zero(),mod(N,succ(succ(zero())))))
     eval(is_pair(N),R) <=>
            eval(is_equal(zero(),mod(N,succ(succ(zero())))),R)
     eval(is_pair(N),R) <=
            eval(is_equal(zero(),mod(N,succ(succ(zero())))),R)
     eval(is_pair(N),R) :- eval(is_equal(0,mod(N,2)),R).
     is_pair(N,R) :- mod(N,2,R1), is_equal(0,R1,R).
```

fonction is_impair(N) donne Br.

Avec N : NAT,
Br : BOOL.

Cette fonction donne un élément de type booléen Br qui est vrai si N est impair (reste de la division de N par 2 différente de 0), faux sinon.

Axiomes_obtenus

```
is_impair(zero()) == false()
is_impair(succ(zero()) == true()
is_impair(succ(N1)) == not(is_impair(N1))

==(is_impair(zero()),false())
==(is_impair(succ(zero()),true())
==(is_impair(succ(N1)),not(is_impair(N1)))

eval(is_impair(zero()),R) <=> R=false()
eval(is_impair(succ(zero()),R) <=> R=true()
eval(is_impair(succ(N1)),R) <=> eval(not(is_impair(N1)),R)

eval(is_impair(N,R) <= eval(N,zero()) & R=false()
eval(is_impair(N,R) <= eval(N,succ(zero()) & R=true()
eval(is_impair(N,R) <= eval(N,succ(N1)) &
                       eval(not(is_impair(N1)),R)

eval(is_impair(N,false) :- eval(N,0).
eval(is_impair(N,true) :- eval(N,N1), N3 is N1-1, N3=0.
eval(is_impair(N,R) :- eval(N,N3), N1 is N3-1,
                       eval(not(is_impair(N1)),R).

is_impair(0,false).
is_impair(1,true).
is_impair(N,R) :- N1 is N-1, is_impair(N1,R2), not(R2,R).

R.C. is_impair(N) == not(is_pair(N))
     ==(is_impair(N),not(is_pair(N)))
     eval(is_impair(N),R) <=> eval(not(is_pair(N)),R)
     eval(is_impair(N),R) <= eval(not(is_pair(N)),R)
     eval(is_impair(N),R) :- eval(not(is_pair(N)),R).
     is_impair(N,R) :- is_pair(N,R1), not(R1,R).
```

Récapitulatif des axiomes ADTs obtenus

```
pred(zero())      == zero()
pred(succ(N1))    == N1

add(N,zero())     == N
add(N,succ(N1))   == succ(add(N,N1))

soust(N,zero())      == N
soust(N,succ(N1))    == pred(soust(N,N1))

mult(N,zero())    == zero()
mult(N,succ(N1)) == add(N,mult(N,N1))

div(N,zero())  == ERREUR "Division par zero illégale"
div(zero(),N2) == zero()
if eval(is_less(succ(N1),N2),true()) then
        div(succ(N1),N2)== zero()
if eval(is_less(succ(N1),N2),false()) then
        div(succ(N1),N2)== succ(div(soust(succ(N1),N2),N2)))

mod(N,zero())  == ERREUR "Division par zero illégale"
mod(zero(),N2) == zero()
if eval(is_less(succ(N1),N2),true()) then
        mod(succ(N1),N2) == succ(N1)
if eval(is_less(succ(N1),N2),false()) then
        mod(succ(N1),N2) == mod(soust(succ(N1),N2),N2)

is_equal(zero(),zero()) == true()
is_equal(zero(),succ(N3)) == false()
is_equal(succ(N1),zero()) == false()
is_equal(succ(N1),succ(N3)) == is_equal(N1,N3)

is_less(zero(),N2) == true()
is_less(succ(N1),zero()) == false()
is_less(succ(N1),succ(N3)) == is_less(N1,N3)

is_leq(zero(),N2)          == true()
is_leq(N,zero())           == is_equal(N,zero())
is_leq(succ(N1),succ(N3)) == is_leq(N1,N3)

is_great(zero(),N2)          == false()
is_great(succ(N1),zero())    == true()
is_great(succ(N1),succ(N3)) == is_great(N1,N3)

is_geq(zero(),N2)          == is_equal(zero(),N2)
is_geq(N,zero())           == true()
is_geq(succ(N1),succ(N3)) == is_geq(N1,N3)

is_diff(zero(),zero()) == false()
is_diff(zero(),succ(N3)) == true()
is_diff(succ(N1),zero()) == true()
is_diff(succ(N1),succ(N3)) == is_equal(N1,N3)

R.C. is_diff(N,N2) == not(is_equal(N,N2))
```

```
      is_pair(zero()) == true()
      is_pair(succ(zero()) == false()
      is_pair(succ(N1)) == not(is_pair(N1))

R.C.  is_pair(N) == is_equal(zero(),mod(N,succ(succ(zero()))))

      is_impair(zero()) == false()
      is_impair(succ(zero()) == true()
      is_impair(succ(N1)) == not(is_impair(N1))

R.C.  is_impair(N) == not(is_pair(N))
```

Récapitulatif des procédures PROLOG avec évaluateur

```
pred(N)

      eval(pred(N),0) :- eval(N,0).
      eval(pred(N),Nr) :- eval(N,N2), Nr is N2-1.

add(N,N2)

      eval(add(N,N2),Nr) :- eval(N2,0),eval(N,Nr).
      eval(add(N,N2),Nr) :- eval(N2,N3), N1 is N3-1,
                            eval(add(N,N1),N3), Nr is N3+1.

soust(N,N2)

      eval(soust(N,N2),Nr) :- eval(N2,0), eval(N,Nr).
      eval(soust(N,N2),Nr) :- eval(N2,N3), N1 is N3-1,
                              eval(pred(soust(N,N1)),Nr).

mult(N,N2)

      eval(mult(N,N2),0) :- eval(N2,0).
      eval(mult(N,N2),Nr) :- eval(N2,N3), N1 is N3-1,
                             eval(add(N,mult(N,N1)),Nr).

div(N,N2)

      eval(div(N,N2),0) :- eval(N,0).
      eval(div(N,N2),0) :- eval(is_less(N,N2),true).
      eval(div(N,N2),Nr) :- eval(is_less(N,N2),false),
                        eval(div(soust(N,N2),N2),N3), Nr is N3+1.

mod(N,N2)

      eval(mod(N,N2),0) :- eval(N,0).
      eval(mod(N,N2),Nr) :- eval(is_less(N,N2),true),
                            eval(N,Nr).
      eval(mod(N,N2),Nr) :- eval(is_less(N,N2),false),
                            eval(mod(soust(N,N2),N2),Nr).
```

```
is_equal(N,N2)

     eval(is_equal(N,N2),true) :- eval(N,0), eval(N2,0).
     eval(is_equal(N,N2),false) :- eval(N,0).
     eval(is_equal(N,N2),false) :- eval(N2,0).
     eval(is_equal(N,N2),Br) :- eval(N,N4), N1 is N4-1,
               eval(N2,N5), N3 is N5-1, eval(is_equal(N1,N3),Br).

is_less(N,N2)

     eval(is_less(N,N2),true) :- eval(N,0).
     eval(is_less(N,N2),false) :- eval(N2,0).
     eval(is_less(N,N2),R) :- eval(N,N4), N1 is N4 - 1, eval(N2,N5),
                              N3 is N5 - 1, eval(is_less(N1,N3),R).

is_leq(N,N2)

     eval(is_leq(N,N2),true) :- eval(N,0).
     eval(is_leq(N,N2),R) :- eval(N2,0),eval(is_equal(N,0),R).
     eval(is_leq(N,N2),R) :- eval(N,N4), N1 is N4 - 1, eval(N2,N5),
                              N3 is N5 - 1, eval(is_leq(N1,N3),R).

is_great(N,N2)

     eval(is_great(N,N2),false) :- eval(N,0).
     eval(is_great(N,N2),true) :- eval(N2,0).
     eval(is_great(N,N2),R) :- eval(N,N4), N1 is N4-1, eval(N2,N5),
                               N3 is N5-1, eval(is_great(N1,N3),R).

is_geq(N,N2)

     eval(is_geq(N,N2),R) :- eval(N,0), eval(is_equal(0,N2),R).
     eval(is_geq(N,N2),true) :- eval(N2,0).
     eval(is_geq(N,N2),R) :- eval(N,N4), N1 is N4-1, eval(N2,N5),
                               N3 is N5-1, eval(is_geq(N1,N3),R).

is_diff(N,N2)

     eval(is_diff(N,N2),false) :- eval(N,0), eval(N2,0).
     eval(is_diff(N,N2),true) :- eval(N,0).
     eval(is_diff(N,N2),true) :- eval(N2,0).
     eval(is_diff(N,N2),Br) :- eval(N,N4), N1 is N4-1,
               eval(N2,N5), N3 is N5-1, eval(is_equal(N1,N3),Br).

is_pair(N)

     eval(is_pair(N,true) :- eval(N,0).
     eval(is_pair(N,false) :- eval(N,N1), N3 is N1-1, N3=0.
     eval(is_pair(N,R) :- eval(N,N3), N1 is N3-1,
                          eval(not(is_pair(N1)),R).

R.C. eval(is_pair(N),R) :- eval(is_equal(0,mod(N,2)),R).
```

```
is_impair(N)

     eval(is_impair(N,false) :- eval(N,0).
     eval(is_impair(N,true) :- eval(N,N1), N3 is N1-1, N3=0.
     eval(is_impair(N,R) :- eval(N,N3), N1 is N3-1,
                            eval(not(is_impair(N1)),R).

R.C. eval(is_impair(N),R) :- eval(not(is_pair(N)),R).
```

Récapitulatif des procédures PROLOG sans évaluateur

```
     pred(0,0).
     pred(N,Nr) :-  Nr is N-1.

     add(N,0,N).
     add(N,N2,Nr) :- N1 is N2-1, add(N,N1,N3), Nr is N3+1.

     soust(N,0,N).
     soust(N,N2,Nr) :- N1 is N2-1, soust(N,N1,N3), pred(N3,Nr).

     mult(0,N,0).
     mult(N,0,0).
     mult(N,N2,Nr) :- N1 is N2-1, mult(N,N1,N3), add(N,N3,Nr).

     div(0,N2,0).
     div(N,N2,0) :- is_less(N,N2).
     div(N,N2,Nr) :- not(is_less(N,N2)),soust(N,N2,N4),
                     div(N4,N2,N3), Nr is N3+1.

     mod(0,N2,0).
     mod(N,N2,N) :- is_less(N,N2).
     mod(N,N2,Nr) :- not(is_less(N,N2)),soust(N,N2,N4),
                     mod(N4,N2,Nr).

     is_equal(0,0,true).
     is_equal(0,N2,false).
     is_equal(N,0,false).
     is_equal(N,N2,Br) :- N1 is N-1, N3 is N2-1, is_equal(N1,N3,Br).

     is_less(0,N2,true).
     is_less(N,0,false).
     is_less(N,N2,Br) :- N1 is N-1, N3 is N2-1,is_less(N1,N3,Br).

     is_leq(0,N2,true).
     is_leq(N,0,R)    :- is_equal(N,0,R).
     is_leq(N,N2,Br) :- N1 is N-1, N3 is N2-1,is_leq(N1,N3,Br).

     is_great(0,N2,false).
     is_great(N,0,true).
     is_great(N,N2,R) :- N1 is N-1, N3 is N2-1, is_great(N1,N3,R).

     is_geq(0,N2,R) :- is_equal(0,N2,R).
     is_geq(N,0,true).
     is_geq(N,N2,R) :- N1 is N-1, N3 is N2-1, is_geq(N1,N3,R).
```

```
     is_diff(0,0,false).
     is_diff(0,N2,true).
     is_diff(N,0,true).
     is_diff(N,N2,Br) :- N1 is N-1, N3 is N2-1,
                         is_equal(N1,N3,Br).

R.C. is_diff(N,N2,R) :- is_equal(N,N2,R2), not(R2,R).

     is_pair(0,true).
     is_pair(1,false).
     is_pair(N,R) :- N1 is N-1, is_pair(N1,R2), not(R2,R).

R.C. is_pair(N,R) :- mod(N,2,R1), is_equal(0,R1,R).

     is_impair(0,false).
     is_impair(1,true).
     is_impair(N,R) :- N1 is N-1, is_impair(N1,R2), not(R2,R).

R.C. is_impair(N,R) :- is_pair(N,R1), not(R1,R).
```

Procédures PROLOG sans évaluateur, sous forme non relationnelles exprimant les foncteurs retournant une valeur booléenne

---

```
     is_equal(0,0).
     is_equal(N,N2) :- N1 is N-1, N3 is N2-1, is_equal(N1,N3).

     is_less(0,N2).
     is_less(N,N2) :- N1 is N-1, N3 is N2-1,is_less(N1,N3).

     is_leq(0,N2).
     is_leq(N,0)   :- is_equal(N,0).
     is_leq(N,N2) :- N1 is N-1, N3 is N2-1,is_leq(N1,N3).

     is_great(N,0).
     is_great(N,N2) :- N1 is N-1, N3 is N2-1, is_great(N1,N3).

     is_geq(0,N2) :- is_equal(0,N2).
     is_geq(N,0).
     is_geq(N,N2) :- N1 is N-1, N3 is N2-1, is_geq(N1,N3).


     is_diff(0,N2) :- N2 != 0.
     is_diff(N,0)  :- N != 0.
     is_diff(N,N2) :- N1 is N-1, N3 is N2-1,
                         is_equal(N1,N3).

R.C. is_diff(N,N2) :- is_equal(N,N2), !, fail.

     is_pair(0).
     is_pair(N) :- N1 is N-1, is_pair(N1), !, fail.

R.C. is_pair(N) :- mod(N,2,R1), is_equal(0,R1).

     is_impair(1).
     is_impair(N) :- N1 is N-1, is_impair(N1), !, fail.

R.C. is_impair(N) :- is_pair(N), !, fail.
```

MODULE ENTIER

TYPE DE DONNEES : ENTIER (integer)

Non paramétré

Inf : Un objet du type ENTIER est un élément dont la valeur appartient à l'ensemble mathématique Z, c'est-à-dire compris entre -∞ et +∞ (infini), par pas de 1 unité.

Set : INT, BOOL, REAL.

Sigma :

Cons	: zero	:	--›	INT
	succ	: INT	--›	INT
	opp	: INT	--›	INT
Modif	: tozero	: INT	--›	INT
	pred	: INT	--›	INT
	abs	: INT	--›	INT
	add	: INT,INT	--›	INT
	soust	: INT,INT	--›	INT
	mult	: INT,INT	--›	INT
	div	: INT,INT	--›	INT
	mod	: INT,INT	--›	INT
	exp	: INT,INT	--›	REAL
	sqr	: INT	--›	INT
Sélec	: is_less	: INT,INT	--›	BOOL
	is_great	: INT,INT	--›	BOOL
	is_eq	: INT,INT	--›	BOOL
	is_leq	: INT,INT	--›	BOOL
	is_geq	: INT,INT	--›	BOOL
	is_diff	: INT,INT	--›	BOOL
	is_pair	: INT,INT	--›	BOOL
	is_impair	: INT,INT	--›	BOOL
	inv	: INT	--›	REAL

Explicatif du rôle des fonctions

zero : Fournit la valeur entière nulle
succ : Fournit la valeur entière suivante d'une autre
opp : Fournit la valeur entière de signe contraire d'une autre

tozero : Fournit la valeur entière précédente d'une valeur entière positive ou suivante d'une valeur entière oppative (valeur rapprochée de zéro).
pred : Fournit la valeur entière précédente d'une autre
abs : Fournit la valeur entière de signe positif si la valeur donnée est oppative
add : Fournit la valeur entière résultant de l'addition de deux autres
soust : Fournit la valeur entière résultant de la soustraction de deux autres
mult : Fournit la valeur entière résultant de la multiplication de deux autres
div : Fournit la valeur entière résultant de la division de deux autres
mod : Fournit la valeur entière reste de la division de deux autres
exp : Fournit la valeur entière égale à l'exposant I2 de I1 (valeurs entières)
sqr : Fournit la valeur entière égale au carré d'une autre

is_less : Détermine si une valeur entière est plus petite qu'une autre
is_great : Détermine si une valeur entière est plus grande qu'une autre
is_eq : Détermine si une valeur entière est égale à une autre
is_leq : Détermine si une valeur entière est plus petite ou égale à une autre
is_geq : Détermine si une valeur entière est plus grande ou égale à une autre
is_diff : Détermine si une valeur entière est différente d'une autre
is_pair : Détermine si une valeur entière est paire (multiple de 2, succ(succ(zero)))
is_impair : Détermine si une valeur entière est impaire (pas multiple de 2, succ(succ(zero)))

inv : Fournit la valeur réelle correspondant à la division de 1 avec une valeur entière donnée

fonction zero() donne Ir .

Avec Ir : INT.

Cette fonction donne un élément de type ENTIER Ir qui prend la valeur nulle zero (Ir=0).

Implémentation : zero() ›prolog› 0

```
eval(zero(),0).

zero(0).
```

fonction succ(I) donne Ir .

Avec I,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui prend la valeur suivante de I dans l'ensemble mathématique Z (Ir=I+1).

Implémentation : succ(I) ›prolog› I + 1

```
eval(succ(I),Ir) :- eval(I,I1), Ir is I1+1.

succ(I,Ir) :- Ir is I+1.
```

fonction opp(I) donne Ir .

Avec I,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui prend la valeur de signe coItraire de I dans l'ensemble mathématique Z (Ir = 0 - I).

Implémentation : opp(I) ›prolog› 0 - I

```
eval(opp(I),Ir) :- eval(I,I1), Ir is 0 - I1.

opp(I,Ir) :- Ir is 0 - I.
```

fonction tozero(I) donne Ir.

Avec I,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui prend la valeur la plus proche de zéro, à partir de I.

Axiomes obtenus

```
tozero(zero())     == zero()
tozero(succ(I1))   == I1
tozero(opp(I1))    == opp(tozero(I1))

==(tozero(zero()),zero())
==(tozero(succ(I1)),I1)
==(tozero(opp(I1)),opp(tozero(I1)))

eval(tozero(zero()),Ir) <=> Ir=zero()
eval(tozero(succ(I1)),Ir) <=> eval(I1,Ir)
eval(tozero(opp(I1)),Ir) <=> eval(opp(tozero(I1)),Ir)

eval(tozero(I),Ir) <= eval(I,zero()) & Ir=zero()
eval(tozero(I),Ir) <= eval(I,succ(I1)) & eval(I1,Ir)
eval(tozero(I),Ir) <= eval(I,opp(I1)) &
                        eval(opp(tozero(I1)),Ir)

eval(tozero(I),0) :- eval(I,0).
eval(tozero(I),Ir) :- eval(I,I2), Ir is I2-1.
eval(tozero(I),Ir) :- eval(I,I2), I1 is 0-I2,
                        eval(tozero(I1),I2), Ir is 0-I2.

tozero(0,0).
tozero(I,Ir) :-  Ir is I-1.
tozero(I,Ir) :-  I1 is 0-I, tozero(I1,I2), Ir is 0-I2.
```

fonction pred(I) donne Ir.

Avec I,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui prend la valeur précédente de I (I - 1).

Axiomes obtenus

```
pred(zero())     == opp(succ(zero())
pred(succ(I1))   == I1
pred(opp(I1))    == opp(succ(I1))
```

```
==(pred(zero()),opp(succ(zero())))
==(pred(succ(I1)),I1)
==(pred(opp(I1)),opp(succ(I1)))

eval(pred(zero()),Ir) <=> Ir=opp(succ(zero())
eval(pred(succ(I1)),Ir) <=> eval(I1,Ir)
eval(pred(opp(I1)),Ir) <=> eval(I1,I2), Ir=opp(succ(I2))

eval(pred(I),Ir) <= eval(I,zero()) & Ir=opp(succ(zero()))
eval(pred(I),Ir) <= eval(I,succ(I1)) & eval(I1,Ir)
eval(pred(I),Ir) <= eval(I,opp(I1)) & Ir=opp(succ(I1))

eval(pred(I),-1) :- eval(I,0).
eval(pred(I),Ir) :- eval(I,I2), Ir is I2-1.
eval(pred(I),Ir) :- eval(I,I2), I1 is 0-I2, Ir is -(I1+1).

pred(0,-1).
pred(I,Ir) :-  Ir is I-1.
pred(I,Ir) :-  I1 is 0-I, Ir is -(I1+1).
```

fonction add(I,I2) donne Ir.

Avec I,I2,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui est le résultat de l'addition de I et I2.

Axiomes obtenus

```
add(zero(),I2)    == I2
add(succ(I1),I2)  == succ(add(I1,I2))
add(opp(I1),I2)   == opp(add(I1,opp(I2))

==(add(zero(),I2),I2)
==(add(succ(I1),I2),succ(add(I1,I2)))
==(add(opp(I1),I2),opp(add(I1,opp(I2)))

eval(add(zero(),I2),Ir) <=> eval(I2,Ir)
eval(add(succ(I1),I2),Ir) <=> eval(succ(add(I1,I2)),Ir)
eval(add(opp(I1),I2),Ir) <=> eval(opp(add(I1,opp(I2))),Ir)

eval(add(I,I2),Ir) <= eval(I,zero()) & eval(I2,Ir)
eval(add(I,I2),Ir) <= eval(I,succ(I1)) &
                      eval(succ(add(I1,I2)),Ir)
eval(add(I,I2),Ir) <= eval(I,opp(I1)) &
                      eval(opp(add(I1,opp(I2))),Ir)

eval(add(I,I2),Ir) :- eval(I,0),eval(I2,Ir).
eval(add(I,I2),Ir) :- eval(I,I3), I1 is I3-1,
                      eval(succ(add(I1,I2)),Ir)
eval(add(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                      eval(opp(add(I1,opp(I2))),Ir).
```

```
eval(add(I,I2),Ir) :- eval(I,0),eval(I2,Ir).
eval(add(I,I2),Ir) :- eval(I,I3), I1 is I3-1,
                      eval(add(I1,I2),I3), Ir is I3+1.
eval(add(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                      eval(add(I1,opp(I2)),I4), Ir is 0-I4.

add(I,0,I).
add(I,I2,Ir) :- I1 is I-1, add(I,I1,I3), Ir is I3+1.
add(I,I2,Ir) :- I1 is 0-I, I3 is 0-I2, add(I1,I2,I4),
                Ir is 0-I4.
```

fonction soust(I,I2) donne Ir.

Avec I,I2,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui est le résultat de la soustraction de I2 à I.

Axiomes_obtenus

```
soust(zero(),I2)      == opp(I2)
soust(succ(I1),I2)    == succ(soust(I1,I2))
soust(opp(I1),I2)     == opp(add(I1,I2))

==(soust(zero(),I2),opp(I2))
==(soust(succ(I1),I2),succ(soust(I1,I2)))
==(soust(opp(I1),I2),opp(add(I1,I2)))

eval(soust(zero(),I2),Ir) <=> eval(opp(I2),Ir)
eval(soust(succ(I1),I2),Ir) <=> eval(succ(soust(I1,I2)),Ir)
eval(soust(opp(I1),I2),Ir) <=> eval(opp(add(I1,I2)),Ir)

eval(soust(I,I2),Ir) <= eval(I,zero()) & eval(opp(I2),Ir)
eval(soust(I,I2),Ir) <= eval(I,succ(I1)) &
                        eval(succ(soust(I1,I2)),Ir)
eval(soust(I,I2),Ir) <= eval(I,opp(I1)) &
                        eval(opp(add(I1,I2)),Ir)

eval(soust(I,I2),Ir) :- eval(I,0), eval(I2,I3), Ir is 0-I3.
eval(soust(I,I2),Ir) :- eval(I,I3), I1 is I3-1,
                        eval(soust(I1,I2),I4), Ir is I4+1.
eval(soust(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                        eval(add(I1,I2),I4), Ir is 0-I4.

soust(0,I2,Ir) :- Ir is 0-I2.
soust(I,I2,Ir) :- I1 is I-1, soust(I1,I2,I4), Ir is I4+1.
soust(I,I2,Ir) :- I1 is 0-I, add(I1,I2,I4), Ir is 0-I4.
```

fonction mult(I,I2) donne Ir.

Avec I,I2,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui est le résultat de la multiplication de I et I2.

Axiomes_obtenus

```
mult(zero(),I2)    == zero()
mult(succ(I1),I2)  == add(I2,mult(I1,I2))
mult(opp(I1),I2)   == opp(mult(I1,I2))

==(mult(zero(),I2),zero())
==(mult(succ(I1),I2),add(I2,mult(I1,I2)))
==(mult(opp(I1),I2),opp(mult(I1,I2)))

eval(mult(zero(),I2),Ir) <=> Ir=zero()
eval(mult(succ(I1),I2),Ir) <=> eval(add(I2,mult(I1,I2)),Ir)
eval(mult(opp(I1),I2),Ir) <=> eval(opp(mult(I1,I2)),Ir)

eval(mult(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(mult(I,I2),Ir) <= eval(I,succ(I1)) &
                       eval(add(I2,mult(I1,I2)),Ir)
eval(mult(I,I2),Ir) <= eval(I,opp(I1)) &
                       eval(opp(mult(I1,I2)),Ir)

eval(mult(I,I2),Ir) :- eval(I,0), Ir=0.
eval(mult(I,I2),Ir) :- eval(I,I3), I1 is I3-1,
                       eval(add(I2,mult(I1,I2)),Ir).
eval(mult(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                       eval(opp(mult(I1,I2)),Ir).

eval(mult(I,I2),Ir) :- eval(I,0), Ir=0.
eval(mult(I,I2),Ir) :- eval(I,I3), I1 is I3-1,
                       eval(add(I2,mult(I1,I2)),Ir).
eval(mult(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                       eval(mult(I1,I2),I4), Ir is 0-I4.

mult(0,I2,0).
mult(I,I2,Ir) :- I1 is I-1, mult(I1,I2,I3), add(I2,I3,Ir).
mult(I,I2,Ir) :- I1 is 0-I, mult(I1,I2,I4), Ir is 0-I4.
```

fonction div(I,I2) donne Ir.

Avec I,I2,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui est le résultat Entier de la division de I par I2.

Axiomes_obtenus

```
div(I,zero())  == ERREUR "Division par zero illégale"
div(zero(),I2) == zero()
div(opp(I1),I2) == opp(div(I1,I2))
if eval(is_less(succ(I1),I2),true()) then
        div(succ(I1),I2)== zero()
if eval(is_less(succ(I1),I2),false()) then
        div(succ(I1),I2)== succ(div(soust(succ(I1),I2),I2)))

div(zero(),I2) == zero()
div(opp(I1),I2) == opp(div(I1,I2))
if eval(is_less(I,I2),true()) then
          div(I,I2)== zero()
if eval(is_less(I,I2),false()) then
          div(I,I2)== succ(div(soust(I,I2),I2))

==(div(zero(),I2),zero())
==(div(opp(I1),I2),opp(div(I1,I2)))
if eval(is_less(I,I2),true()) then
          ==(div(I,I2),zero())
if eval(is_less(I,I2),false()) then
          ==(div(I,I2),succ(div(soust(I,I2),I2)))

eval(div(zero(),I2),Ir) <=> Ir=zero()
eval(div(opp(I1),I2),Ir) <=> eval(opp(div(I1,I2)),Ir)
if eval(is_less(I,I2),true()) then
   eval(div(I,I2),Ir) <=> Ir=zero()
if eval(is_less(I,I2),false()) then
   eval(div(I,I2),Ir) <=> eval(succ(div(soust(I,I2),I2)),Ir)

eval(div(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(div(I,I2),Ir) <= eval(I,opp(I1)) &
                      eval(opp(div(I1,I2)),Ir)
if eval(is_less(I,I2),true()) then
   eval(div(I,I2),Ir) <= Ir=zero()
if eval(is_less(I,I2),false()) then
   eval(div(I,I2),Ir) <= eval(succ(div(soust(I,I2),I2)),Ir)

eval(div(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(div(I,I2),Ir) <= eval(I,opp(I1)) &
                      eval(opp(div(I1,I2)),Ir)
(eval(div(I,I2),Ir) <= Ir=zero())
                   <= eval(is_less(I,I2),true())
(eval(div(I,I2),Ir) <= eval(succ(div(soust(I,I2),I2)),Ir))
                   <= eval(is_less(I,I2),false())
```

```
eval(div(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(div(I,I2),Ir) <= eval(I,opp(I1)) &
                      eval(opp(div(I1,I2)),Ir)
eval(div(I,I2),Ir) <= eval(is_less(I,I2),true())
                       & Ir=zero())
eval(div(I,I2),Ir) <= eval(is_less(I,I2),false())
                & eval(succ(div(soust(I,I2),I2)),Ir)

eval(div(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(div(I,I2),Ir) <= eval(I,opp(I1)) &
                      eval(opp(div(I1,I2)),Ir)
eval(div(I,I2),Ir) <= eval(is_less(I,I2),true())
                       & Ir=zero())
eval(div(I,I2),Ir) <= eval(is_less(I,I2),false())
                & eval(succ(div(soust(I,I2),I2)),Ir)

eval(div(I,I2),0) :- eval(I,0).
eval(div(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                      eval(opp(div(I1,I2)),Ir).
eval(div(I,I2),0) :- eval(is_less(I,I2),true).
eval(div(I,I2),Ir) :- eval(is_less(I,I2),false),
                      eval(succ(div(soust(I,I2),I2)),Ir).

eval(div(I,I2),0) :- eval(I,0).
eval(div(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                      eval(div(I1,I2),I4), Ir is 0-I4.
eval(div(I,I2),0) :- eval(is_less(I,I2),true).
eval(div(I,I2),Ir) :- eval(is_less(I,I2),false),
                   eval(div(soust(I,I2),I2),I3), Ir is I3+1.

div(0,I2,0).
div(I,I2,Ir) :- I1 is 0-I, div(I1,I2,I4), Ir is 0-I4.
div(I,I2,0) :- is_less(I,I2,true).
div(I,I2,Ir) :- is_less(I,I2,false),soust(I,I2,I4),
                div(I4,I2,I3), Ir is I3+1.

div(0,I2,0).
div(I,I2,Ir) :- I1 is 0-I, div(I1,I2,I4), Ir is 0-I4.
div(I,I2,0) :- is_less(I,I2).
div(I,I2,Ir) :- Iot(is_less(I,I2)), soust(I,I2,I4),
                div(I4,I2,I3), Ir is I3+1.
```

fonction mod(I,I2) donne Ir.

Avec I,I2,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui est le reste de la division de I par I2.

Axiomes_obtenus

```
mod(I,zero())  == ERREUR "Division par zero illégale"
mod(zero(),I2) == zero()
mod(opp(I1),I2) == mod(I1,I2)
if eval(is_less(succ(I1),I2),true()) then
        mod(succ(I1),I2) == succ(I1)
if eval(is_less(succ(I1),I2),false()) then
        mod(succ(I1),I2) == mod(soust(succ(I1),I2),I2)

mod(zero(),I2) == zero()
mod(opp(I1),I2) == mod(I1,I2)
if eval(is_less(I,I2),true()) then
          mod(I,I2) == I
if eval(is_less(I,I2),false()) then
          mod(I,I2) == mod(soust(I,I2),I2)

==(mod(zero(),I2),zero())
==(mod(opp(I1),I2), mod(I1,I2))
if eval(is_less(I,I2),true()) then
          ==(mod(I,I2),I)
if eval(is_less(I,I2),false()) then
          ==(mod(I,I2),mod(soust(I,I2),I2))

eval(mod(zero(),I2),Ir) <=> Ir=zero()
eval(mod(opp(I1),I2),Ir) <=> eval(mod(I1,I2),Ir)
if eval(is_less(I,I2),true()) then
         eval(mod(I,I2),Ir) <=> eval(I,Ir)
if eval(is_less(I,I2),false()) then
         eval(mod(I,I2),Ir) <=> eval(mod(soust(I,I2),I2),Ir)

eval(mod(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(mod(I,I2),Ir) <= eval(I,opp(I1)) & eval(mod(I1,I2),Ir)
if eval(is_less(I,I2),true()) then
         eval(mod(I,I2),Ir) <= eval(I,Ir)
if eval(is_less(I,I2),false()) then
         eval(mod(I,I2),Ir) <= eval(mod(soust(I,I2),I2),Ir)

eval(mod(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(mod(I,I2),Ir) <= eval(I,opp(I1)) & eval(mod(I1,I2),Ir)
(eval(mod(I,I2),Ir) <= eval(I,Ir)) <=
                    eval(is_less(I,I2),true())
(eval(mod(I,I2),Ir) <= eval(mod(soust(I,I2),I2),Ir)) <=
                    eval(is_less(I,I2),false())
```

```
eval(mod(I,I2),Ir) <= eval(I,zero()) & Ir=zero()
eval(mod(I,I2),Ir) <= eval(I,opp(I1)) & eval(mod(I1,I2),Ir)
eval(mod(I,I2),Ir) <= eval(is_less(I,I2),true())
                      & eval(I,Ir)
eval(mod(I,I2),Ir) <= eval(is_less(I,I2),false()) &
                      eval(mod(soust(I,I2),I2),Ir)

eval(mod(I,I2),0) :- eval(I,0).
eval(mod(I,I2),Ir) :- eval(I,I3), I1 is 0-I3,
                      eval(mod(I1,I2),Ir).
eval(mod(I,I2),Ir) :- eval(is_less(I,I2),true),
                      eval(I,Ir).
eval(mod(I,I2),Ir) :- eval(is_less(I,I2),false),
                      eval(mod(soust(I,I2),I2),Ir).

mod(0,I2,0).
mod(I,I2,Ir) :- I1 is 0-I, mod(I1,I2,Ir).
mod(I,I2,I) :- is_less(I,I2,true).
mod(I,I2,Ir) :- is_less(I,I2,false),soust(I,I2,I4),
                mod(I4,I2,Ir).

mod(0,I2,0).
mod(I,I2,Ir) :- I1 is 0-I, mod(I1,I2,Ir).
mod(I,I2,I) :- is_less(I,I2).
mod(I,I2,Ir) :- Iot(is_less(I,I2)),soust(I,I2,I4),
                mod(I4,I2,Ir).
```

fonction abs(I) donne Ir.

Avec I,Ir : INT.

Cette fonction donne un élément de type Entier Ir qui prend la valeur absolue (positive) de l'entier I

Axiomes_obtenus

```
abs(zero()) == zero()
abs(succ(I1)) == succ(abs(I1))
abs(opp(I1)) == abs(I1)

==(abs(zero()),zero())
==(abs(succ(I1)),succ(abs(I1)))
==(abs(opp(I1)),abs(I1))

eval(abs(zero()),Ir) <=> Ir=zero()
eval(abs(succ(I1),Ir) <=> eval(succ(abs(I1)),Ir)
eval(abs(opp(I1)),Ir) <=> eval(abs(I1),Ir)

eval(abs(I),Ir) <= eval(I,zero()) & Ir=zero()
eval(abs(I),Ir) <= eval(I,succ(I1)) & eval(succ(abs(I1)),Ir)
eval(abs(I),Ir) <= eval(I,opp(I1)) & eval(abs(I1),Ir)
```

```
eval(abs(I),0) :- eval(I,0).
eval(abs(I),Ir) :- eval(I,I2), I1 is I2-1, eval(abs(I1),I3),
                   Ir is I3+1.
eval(abs(I),Ir) :- eval(I,I2), I1 is 0-I2, eval(abs(I1),Ir).

abs(0,0).
abs(I,Ir) :- I1 is I-1, abs(I1,I3), Ir is I3+1.
abs(I,Ir) :- I1 is 0-I, abs(I1,Ir).
```

fonction exp(I,I2) donne Ir.

Avec I,I2,Ir : INT.

Cette fonction donne un élément de type ENTIER Ir qui est l'entier I à l'exposant I2 en valeur absolue (Ir = I^I2).

Axiomes obtenus

```
exp(I,succ(zero()) == I
exp(I,succ(I1)) == mult(I,exp(I,I1))
exp(I,zero()) == succ(zero())
exp(I,opp(I1)) == exp(I,I1)          Remarque : choix

==(exp(I,succ(zero()),I)
==(exp(I,succ(I1)),mult(I,exp(I,I1)))
==(exp(I,zero()),succ(zero()))
==(exp(I,opp(I1)),exp(I,I1))

eval(exp(I,succ(zero()),Ir) <=> eval(I,Ir)
eval(exp(I,succ(I1)),Ir) <=> eval(mult(I,exp(I,I1)),Ir)
eval(exp(I,zero()),Ir) <=> Ir=succ(zero()))
eval(exp(I,opp(I1)),Ir) <=> eval(exp(I,I1),Ir)

eval(exp(I,I2),Ir) <= eval(I2,succ(zero()) & eval(I,Ir)
eval(exp(I,I2),Ir) <= eval(I2,succ(I1)) &
                      eval(mult(I,exp(I,I1)),Ir)
eval(exp(I,I2),Ir) <= eval(I2,zero()) & Ir=succ(zero()))
eval(exp(I,I2),Ir) <= eval(I2,opp(I1)) & eval(exp(I,I1),Ir)

eval(exp(I,I2),Ir) :- eval(I2,1), eval(I,Ir).
eval(exp(I,I2),Ir) :- eval(I2,I3), I1 is I3-1,
                      eval(mult(I,exp(I,I1)),Ir).
eval(exp(I,I2),1) :- eval(I2,0).
eval(exp(I,I2),Ir) :- eval(I2,I3), I1 is 0-I3,
                      eval(exp(I,I1),Ir).

exp(I,1,I).
exp(I,I2,Ir) :- I1 is I2-1, exp(I,I1,I3), mult(I,I3,Ir).
exp(I,0,1).
exp(I,I2,Ir) :- I1 is 0-I2, exp(I,I1,Ir).
```

MODULE LISTE

TYPE DE DONNEES : LISTE (séquence,suite) N° 1

Paramétré par ELEM

Inf : Un objet du type LISTE est un ensemble ordonné d'éléments de même type. L'ordre concerne les valeurs d'une propriété commune à tous les éléments. La plupart du temps, la propriété implicite est l'ordre temporel d'insertion.

Set : SEQ, NAT, BOOL, ELEM.

Sigma :

Cons	:	empty	:	--› SEQ
		app	: ELEM,SEQ	--› SEQ
Modif	:	conc	: SEQ,SEQ	--› SEQ
		insrt	: ELEM,NAT,SEQ	--› SEQ
		del	: NAT,SEQ	--› SEQ
		efface	: ELEM,SEQ	--› SEQ
		eff_all	: ELEM,SEQ	--› SEQ
		subseq	: NAT,NAT,SEQ	--› SEQ
		tail	: SEQ	--› SEQ
		reverse	: SEQ	--› SEQ
		linear	: SEQ	--› SEQ
		sort	: SEQ,PRED	--› SEQ
		filter	: SEQ,ELEM,PRED	--› SEQ
		merge	: SEQ,SEQ,PRED	--› SEQ
Sélec	:	ith	: NAT,SEQ	--› ELEM
		head	: SEQ	--› ELEM
		last	: SEQ	--› ELEM
		length	: SEQ	--› NAT
		is_in	: ELEM,SEQ	--› BOOL
		is_empty	: SEQ	--› BOOL
		is_sort	: SEQ,PRED	--› BOOL

Explicatif du rôle des fonctions

empty : Création d'une liste vide.
app : Ajout d'un élément en queue de liste.

conc : Concaténation de deux listes.
insrt : Insertion d'un élément à la position i dans une liste.
del : Enlève l'élément à la position I.
efface : Enlève la première occurence de l'élément E dans la liste S.
eff_all : Enlève toutes les occurences de l'élément E dans la liste S.

subseq : Fournit la sous-liste des éléments des positions N1 à N2.
tail : fournit la liste de tous les éléments de la liste argument, excepté le premier.
reverse : Fournit une liste dont l'ordre des éléments est l'inverse de la liste argument.
linear : Fournit la liste des éléments à tous les niveaux de sous-listes de S
sort : Tri d'une liste selon un ordre de valeurs d'une propriété P commune à tous les éléments.
filter : Filtre une liste selon une propriété P commune à tous les éléments.
merge : Fusionne deux listes selon une propriété commune à tous les éléments.

ith : Fournit le i-ème élément de la liste.
head : Fournit l'élément en tête de liste
last : Fournit l'élément en queue de liste

length : Fournit la longueur d'une liste.

is_in : Vérifie l'existence d'un élément dans une liste.
is_empty : Vérifie si la liste est vide.
is_sort : Vérifie si la liste est triée selon une propriété

simple : Applique la fonction OP à chaque élément de la liste.

MODULE LISTE

TYPE DE DONNEES : LISTE (séquence,suite) N° 2

Modif :	correct :	ELEM,NAT,SEQ	--› SEQ
	subst_all:	ELEM,ELEM,SEQ	--› SEQ
	subst_one:	ELEM,ELEM,SEQ	--› SEQ
	subcut :	ELEM,ELEM,SEQ	--› SEQ
	prefcut :	ELEM,SEQ	--› SEQ
	newseq :	ELEM,NAT	--› SEQ
Sélec :	times :	ELEM,SEQ	--› NAT
	pos :	ELEM,SEQ	--› NAT
	is_at :	ELEM,NAT,SEQ	--› BOOL
	is_sublist :	SEQ,SEQ	--› BOOL
	is_prefix:	SEQ,SEQ	--› BOOL
	is_ident :	SEQ,SEQ	--› BOOL

Explicatif du rôle des fonctions

correct : Remplace l'élément à telle position de la séquence par un autre élément.
subst_all : Remplace tous les éléments d'une séquence par un autre élément.
subst_one : Remplace la première occurence d'un élément de la séquence par un autre élément.
subcut : Extrait d'une séquence la sous-séquence comprise entre les premières occurences de deux éléments
prefcut : Extrait d'une séquence la sous-séquence allant du début jusqu'à la première occurence d'un élément.
newseq : Crée une séquence ayant un nombre fixé d'éléments tous initialisés à la même valeur.

times : Détermine le nombre de fois qu'un élément apparaît dans une séquence.
pos : Détermine la position d'un élément d'une séquence
is_at : Détermine si un élément est bien à telle position d'une séquence.
is_sublist : Détermine si une séquence est une sous-séquence d'une autre.
is_prefix : Détermine si une séquence est une sous-séquence initiale (en début) d'une autre.
is_ident : détermine si deux sous-séquences sont identiques.

Elaboration des spécifications sémantiques

fonction empty() donne Sr .

Avec Sr : SEQ.

Cette fonction donne une liste résultante Sr qui est la liste vide.

Postconditions : Sr est la liste vide.

Implémentation

eval(S,empty()) ›prolog› eval(S,[])

ou remplacement direct des occurences de S par []

S=empty() ›prolog› S=[]

fonction app(E,S) donne Sr .

Avec E : ELEM,
S,Sr : SEQ.

Cette fonction donne une liste résultante Sr qui est la liste S à laquelle on a ajouté en tête l'élément E.

Implémentation

eval(S,app(E,S2)) ›prolog› eval(S,[E¦S2])

ou alors remplacement direct des occurences de S par [E¦S2]

S=app(E,S2) ›prolog› S=[E¦S2]

fonction conc(S1,S2) donne Sr .

Avec S1,S2,Sr : SEQ,

Cette fonction donne une liste résultante Sr qui est la concaténation de S2 à S1.

Paramètre d'induction : S1.

S1 ou S2 sont de bons candidats. Nous choisissons S2 en pensant que nous allons ajouter à la liste SR la liste S2 puis chaque élément un à un de S1.

Remarque : ce choix est dû au fait que la spécification de la fonction app est d'ajouter en tête de liste l'élément, donc devant.

Relation bien-fondée : s1<s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

S1 cons1 : empty()
cons2 : app(E,S3)

Construction des Fi

- si S1 est empty(), SR est S2.
- si S1 est app(E,S),
  on ajoute l'élément E à la liste résultant de la concaténation de S avec S2.

Axiomes obtenus

conc(empty(),S2) == S2
conc(app(E,S),S2) == app(E,conc(S,S2))

Cet ensemble d'axiomes est complet, puisque S1 est représenté par tous les constructeurs possibles, et que S2 peut être n'importe quel constructeur de liste.

==(conc(empty(),S2),S2)
==(conc(app(E,S1),S2),app(E,conc(S1,S2)))

eval(conc(empty(),S2),R) <=> eval(S2,R)
eval(conc(app(E,S1),S2),R) <=> eval(app(E,conc(S1,S2)),R)

```
eval(conc(S,S2),R) <= S=empty() & eval(S2,R)
eval(conc(S,S2),R) <= S=app(E,S1) &
                      eval(app(E,conc(S1,S2)),R)

eval(conc(S,S2),R) <= eval(S,S1) & S1=empty() &
                      eval(S2,R)
eval(conc(S,S2),R) <= eval(S,S3) & S3=app(E,S1) &
                      eval(app(E,conc(S1,S2)),R)

eval(conc(S,S2),R):-eval(S,S1),S1=[],eval(S2,R).
eval(conc(S,S2),R) :- eval(S,S3),S3=[E|S1],
                      eval(app(E,conc(S1,S2)),R).

eval(conc(S,S2),R):-eval(S,[]),eval(S2,R).
eval(conc(S,S2),[E|I]) :- eval(S,[E|S1]),
                          eval(conc(S1,S2),I).

eval(conc(S,S2),R)     :- eval(S,[]),eval(S2,R).
eval(conc(S,S2),[E|I]) :- eval(S,[E|S1]),
                          eval(conc(S1,S2),I).

conc([],S2,S2).
conc([E|S1],S2,[E|I]) :- conc(S1,S2,I).
```

```
fonction insrt(E,N,S) donne Sr .

Avec E     : ELEM,
     N     : NAT,
     S,Sr  : SEQ,

Cette fonction donne une liste résultante SR qui est
la liste S avec l'élément E inséré à la Nième position
dans la liste S.

Préconditions  :  N > zero(),N <= leng(S).
```

Paramètre d'induction : N.

E est un mauvais candidat, puisque il peut être d'un type non simple.

S est un bon candidat, de même que N. Mais comme il est parfois nécessaire de connaître la longueur de la liste si on choisit S, on préfère N, qui indique directement la position (profondeur) dans la liste. (En fait, il y a double induction structurelle, à la fois sur N et sur S. Seulement, il faut déterminer l'ordre de choix de l'induction afin de limiter le nombre d'axiomes).

Relation_bien-fondée

n1<n2 ssi n1 a une valeur plus petite que n2.
Elément minimal : succ(zero()). (et pas zero() car précondition)

Forme_structurelle_du_paramètre_d'induction

N cons1 : succ(zero())
cons2 : N > cons1, succ(...(succ(zero()))...)

Construction_des_Fi

- si N est succ(zero()),
dans les deux cas (liste vide ou non), on ajoute l'élément en tête.
- si N est succ(...(succ(zero()))...), ou succ(N),
si S=empty() alors ajout de l'élément en tête, ou message d'erreur ['erreur sur longueur de chaîne']
si S=app(E2,S2) alors
on ajoute l'élément E2 à la liste dans laquelle on aura inséré l'élément E dans la liste queue S2, mais à une position de moins (N).

Axiomes_obtenus

```
insrt(E,succ(zero()),S) == app(E,S)
insrt(E,succ(N),app(E2,S2)) == app(E2,insrt(E,N,S2))

==(insrt(E,succ(zero()),S),app(E,S))
==(insrt(E,succ(N),app(E2,S2)),app(E2,insrt(E,N,S2)))

eval(insrt(E,succ(zero()),S),R) <=> R=app(E,S)
eval(insrt(E,succ(N),app(E2,S2)),R) <=>
                              eval(app(E2,insrt(E,N,S2)),R)

eval(insrt(E,N,S),R) <= eval(N,succ(zero()) &
                        R=app(E,S)
eval(insrt(E,N,S),R) <= eval(S,app(E2,S2)) &
          eval(N,succ(N1)) & eval(app(E2,insrt(E,N,S2)),R)

eval(insrt(E,N,S),[E|S]) :- eval(N,1).
eval(insrt(E,N,S),[E2|R]) :- eval(S,[E2|S2]),eval(N,N1),
             N2 is N1-1, eval(insrt(E,N2,S2),R).

insrt(E,1,S,[E|S]).
insrt(E,N,[E2|S2],[E2|R]) :- N2 is N-1, insrt(E,N2,S2,R).
```

fonction del(N,S) donne Sr .

Avec N : NAT,
S,Sr : SEQ,

Cette fonction donne une liste Sr qui est reprend tous les éléments, excepté le Nième, de la liste S, dans le même ordre

Préconditions : N > 0, N <= leng(S),
is_empty(S)=false()

Paramètre_d'induction : N.

Relation_bien-fondée : n1<n2 ssi n1 a une valeur plus petite que n2

Forme_structurelle_du_paramètre_d'induction

N cons1 : succ(zero())
cons2 : succ(...(succ(zero()))...)

Construction_des_Fi

- si N est succ(zero()), on ne garde que la liste queue, ou la liste vide si S est empty() (on peut mettre un message d'erreur en disant que l'on ne peut pas accepter d'effacer un élément à telle position d'une liste vide.)
- si N est succ(N), on ajoute l'élément E à la liste queue dans laquelle on enlèvera l'élément de la position N.

Axiomes_obtenus

```
del(succ(zero()),app(E,S2)) == S2
del(succ(N),app(E,S2))      == app(E,del(N,S2))

==(del(succ(zero()),app(E,S2)),S2)
==(del(succ(N),app(E,S2)),app(E,del(N,S2)))

eval(del(succ(zero()),app(E,S2)),R) <=> eval(S2,R)
eval(del(succ(N),app(E,S2)),R) <=> eval(app(E,del(N,S2)),R)

eval(del(N,S),R) <= N=succ(zero()) & S=app(E,S2) &
                    eval(S2,R)
eval(del(N,S),R) <= S=app(E,S2) & N=succ(N1)
                    eval(app(E,del(N1,S2)),R)
```

```
eval(del(N,S),R) <= eval(N,N1) & N1=succ(zero()) &
                    eval(S,S1) & S1=app(E,S2) &
                    eval(S2,R)
eval(del(N,S),R) <= eval(S,S1) & S1=app(E,S2) &
                    eval(N,N2) & N2=succ(N1)
                    eval(app(E,del(N1,S2)),R)
```

remarque, ici dessus, le terme S2 à déjà été évalué par eval(S,S1) & S1=app(E,S2). Il n'est donc plus nécessaire de le faire. On peut remplacer les occurences de S2 par R.

```
eval(del(N,S),R) <= eval(N,N1) & N1=succ(zero()) &
                    eval(S,S1) & S1=app(E,R))
eval(del(N,S),R) <= eval(S,S1) & S1=app(E,S2) &
                    eval(N,N2) & N2=succ(N1)
                    eval(del(N1,S2),S3) & R=app(E,S3)
```

```
eval(del(N,S),R)     :- eval(N,1),eval(S,[E|R]),
eval(del(N,S),[E|R]) :- eval(S,[E|S2]),eval(N,N2),
                        N1 is N2-1, eval(del(N1,S2)),R).
```

```
del(1,[_|S2],S2).
del(N,[E|S2],[E|R]) :- N1=N-1,del(N1,S2,R).
```

fonction efface(E,S) donne Sr .

Avec S,Sr : SEQ.
Er : ELEM.

Cette fonction donne une liste Sr qui est la liste des éléments de la liste S, excepté la première occurence de E.

Paramètre d'induction S

Relation bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

S cons1 : empty()
cons2 : app(E,S2)

Construction des Fi

- si S est empty(), il n'y a rien à enlever.
- si S est app(E2,S2). Il est nécessaire de vérifier que l'élément en tête soit égal ou différent de E. En cas d'égalité, on ne retourne que la suite S2, sinon, on ajoute au résultat l'élément en tête, et le reste du résultat est obtenu par récursion sur S2.

axiomes obtenus

```
efface(E,empty())     == empty()
efface(E,app(E,S2))   == S2
efface(E,app(E2,S2)) == app(E2,efface(E,S2))

==(efface(E,empty()),empty())
==(efface(E,app(E,S2)),S2)
==(efface(E,app(E2,S2)),app(E2,efface(E,S2)))

eval(efface(E,empty()),R) <=> R=empty()
eval(efface(E,app(E,S2)),R) <=> eval(S2,R)
eval(efface(E,app(E2,S2)),R) <=>
                        eval(app(E2,efface(E,S2)),R)

eval(efface(E,S),R) <= eval(S,S1) & S1=empty() & R=empty()
eval(efface(E,S),R) <= eval(S,S1) & S1=app(E,S2) &
                        eval(S2,R)
eval(efface(E,S),R) <= eval(S,S1) & S1=app(E2,S2) &
                        eval(app(E2,efface(E,S2)),R)

eval(efface(E,S),[]) :- eval(S,[]).
eval(efface(E,S),R)  :- eval(S,[E|S2]),eval(S2,R).
eval(efface(E,S),[E2|I]) :- eval(S,[E2|S2]),
                            eval(efface(E,S2),I).
```

```
efface(_,[],[]).
efface(E,[E|S2],S2).
efface(E,[E2|S2],[E2|I]) :- efface(E,S2,I).
```

fonction eff_all(E,S) donne Sr .

Avec S,Sr : SEQ.
Er : ELEM.

Cette fonction donne une liste Sr qui est la liste des éléments de la liste S, excepté les occurences de E.

Paramètre d'induction S

Relation bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

S cons1 : empty()
cons2 : app(E,S2)

Construction des Fi

- si S est empty(), il n'y a rien à enlever.
- si S est app(E2,S2). Il est nécessaire de vérifier que l'élément en tête soit égal ou différent de E. En cas d'égalité, le résultat est donné par l'appel récursif sur S2 (on continue à supprimer les E), sinon, on ajoute au résultat l'élément en tête, et le reste du résultat est obtenu par récursion sur S2.

axiomes obtenus

```
eff_all(E,empty())     == empty()
eff_all(E,app(E,S2))   == eff_all(E,S2)
eff_all(E,app(E2,S2)) == app(E2,eff_all(E,S2))

==(eff_all(E,empty()),empty())
==(eff_all(E,app(E,S2)),eff_all(E,S2))
==(eff_all(E,app(E2,S2)),app(E2,eff_all(E,S2)))

eval(eff_all(E,empty()),R) <=> R=empty()
eval(eff_all(E,app(E,S2)),R) <=> eval(eff_all(E,S2),R)
eval(eff_all(E,app(E2,S2)),R) <=>
                    eval(app(E2,eff_all(E,S2)),R)

eval(eff_all(E,S),R) <= eval(S,S1) & S1=empty() & R=empty()
eval(eff_all(E,S),R) <= eval(S,S1) & S1=app(E,S2) &
                        eval(eff_all(E,S2),R)
eval(eff_all(E,S),R) <= eval(S,S1) & S1=app(E2,S2) &
                        eval(app(E2,eff_all(E,S2)),R)
```

```
eval(eff_all(E,S),[]) :- eval(S,[]).
eval(eff_all(E,S),R)  :- eval(S,[E|S2]),
                         eval(eff_all(E,S2),R).
eval(eff_all(E,S),[E2|I]) :- eval(S,[E2|S2]),
                             eval(eff_all(E,S2),I).

eff_all(_,[],[]).
eff_all(E,[E|S2],R) :- eff_all(E,S2,R).
eff_all(E,[E2|S2],[E2|I]) :- eff_all(E,S2,I).
```

fonction subseq(N,N2,S) donne Sr.

Avec N,N2 : NAT,
S,Sr : SEQ,

Cette fonction donne une liste Sr qui est la sous-liste des éléments de la liste S, de la position N à N2.

Préconditions : N ‹= N2, N › 0, N2 ‹= leng(S)
is_empty(S) == false()

Paramètre_d'induction N

Relation_bien-fondée

n1‹n2 si n1 a une valeur plus petite que n2

Forme_structurelle_du_paramètre_d_induction

N cons1 : succ(zero())
cons2 : succ(...(succ(zero()))...)

Construction_des_Fi

- si N est succ(zero()) Ici, on est au début de la sous-liste. l'induction se fera sur le second paramètre N2.

N2 cons1 : succ(zero())
cons2 : succ(...(succ(zero()))...)

- si N2 est succ(zero()), Sr est app(E,empty()).
- si N2 est succ(...(succ(zero()))...), on ajoute l'élément à la sous-liste fournie par l'appel de subseq avec le prédécesseur de N2 .

- si N est succ(...(succ(zero()))...),
dans ce cas, on effectue un appel récursif avec les deux prédécesseurs de N et N2 (pour que N tend vers l'élément minimal), et on ne considère que la queue de la liste. Le résultat de l'appel récursif doit être ajouté à l'élément en tête de liste.

Axiomes_obtenus

```
subseq(succ(zero()),succ(zero()),app(E,S2)) ==
       app(E,empty())
subseq(succ(zero()),N2,app(E,S2)) ==
       app(E,subseq(succ(zero()),pred(N2),S2))
subseq(N,N2,app(E,S2)) == subseq(pred(N),pred(N2),S2)

==(subseq(succ(zero()),succ(zero()),app(E,S2)),
       app(E,empty()))
==(subseq(succ(zero()),N2,app(E,S2)),
       app(E,subseq(succ(zero()),pred(N2),S2)))
==(subseq(N,N2,app(E,S2)),subseq(pred(N),pred(N2),S2))

eval(subseq(succ(zero()),succ(zero()),app(E,S2)),R) <=>
       eval(app(E,empty()),R)
eval(subseq(succ(zero()),N2,app(E,S2)),R) <=>
       eval(app(E,subseq(succ(zero()),pred(N2),S2)),R)
eval(subseq(N,N2,app(E,S2)),R) <=>
       eval(subseq(pred(N),pred(N2),S2),R)

eval(subseq(N,N2,S),R) <= eval(N,N1) & N1=succ(zero()) &
       eval(N2,N3) & N3=succ(zero()) &
       eval(S,S1) & S1=app(E,S2) & eval(app(E,empty()),R)
eval(subseq(N,N2,S),R) <= eval(N,N1) & N1=succ(zero()) &
       eval(S,S1) & S1=app(E,S2) &
       eval(app(E,subseq(succ(zero()),pred(N2),S2)),R)
eval(subseq(N,N2,S),R) <= eval(S,S1) & S1=app(E,S2) &
       eval(subseq(pred(N),pred(N2),S2),R)

eval(subseq(N,N2,S),[E]) :- eval(N,1),eval(N2,1),
                            eval(S,[E|S2].
eval(subseq(N,N2,S),[E|I]) :- eval(N,1),eval(S,[E|S2]),
                          eval(subseq(1,pred(N2),S2),I).
eval(subseq(N,N2,S),R) :- eval(S,[E|S2]),
                     eval(subseq(pred(N),pred(N2),S2),R).

subseq(1,1,[E|_],[E]).
subseq(1,N2,[E|S2],[E|I]) :- N3 is N2-1, subseq(1,N3,S2,I).
subseq(N,N2,[_|S2],R) :- N1 is N-1,N3 is N2-1,
                         subseq(N1,N3,S2,R).
```

fonction tail(S) donne Sr .

Avec S,Sr : SEQ.

Cette fonction donne une liste Sr qui est la liste des éléments de S, sauf le premier.

Préconditions : is_empty(S) == false()

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

On ne peut envisager le cas S=empty selon la précondition.
S cons1 : app(E,S2)

Construction_des_Fi

- si S est app(E,S2), il est évident que la queue de S est S2 (définition du constructeur app).

axiomes_obtenus

tail(app(E,S2)) == S2

==(tail(app(E,S2)),S2)

eval(tail(app(E,S2)),R) ‹=› eval(S2,R)

eval(tail(S),R) ‹= eval(S,S1) & S1=app(E,S2) & eval(S2,R)

eval(tail(S),R) :- eval(S,[E¦S2]),eval(S2,R).

tail([_¦S2],S2).

fonction reverse(S) donne Sr .

Avec S,Sr : SEQ,

Cette fonction donne une liste Sr qui est la liste des éléments de la liste S, repris dans l'ordre inverse.

Paramètre d'induction S

Relation bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

```
S    cons1 : empty()
     cons2 : app(E,S2)
```

Construction des Fi

- si S est empty(), l'inverse d'une liste vide est une liste vide.
- si S est app(E,S2), on concatène la liste inverse de la queue avec la liste ne comportant que l'élément E.

axiomes obtenus

```
reverse(empty())    == empty()
reverse(app(E,S2)) == conc(reverse(S2),app(E,empty))

==(reverse(empty()),empty())
==(reverse(app(E,S2)),conc(reverse(S2),app(E,empty())))

eval(reverse(empty()),R) <=> R=empty()
eval(reverse(app(E,S2)),R) <=>
                  eval(conc(reverse(S2),app(E,empty())),R)

eval(reverse(S),R) <=> eval(S,S1) & S1=empty() & R=empty()
eval(reverse(S),R) <=> eval(S,S1) & S1=app(E,S2) &
                  eval(conc(reverse(S2),app(E,empty())),R)

eval(reverse(S),[]) :- eval(S,[]).
eval(reverse(S),R) :- eval(S,[E|S2]),
                      eval(conc(reverse(S2),[E]),R).

reverse([],[]).
reverse([E|S2],R) :- reverse(S2,I),conc(I,[E],R).
```

fonction linear(S) donne Sr .

Avec S,Sr : SEQ.

Cette fonction donne une liste Sr qui est la liste comportant tous les éléments de la liste S et de tous les niveaux des sous-listes

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

```
S    cons1 : empty()
     cons2 : app(E,S2)
```

Construction_des_Fi

- si S est empty(), Sr est empty().
- si S est app(E,S2), selon que E soit une liste ou non, le traitement diffèrera : E est une liste, alors on concatène la liste obtenue par linéarisation de E, avec celle de S2. Sinon, on concatène la liste formée du seul élément E avec la liste obtenue par linéarisation de S.

axiomes_obtenus

```
linear(empty()) == empty()
if is_list(E) then
   linear(app(E,S2)) == conc(linear(E),linear(S))
if not(is_list(E)) then
   linear(app(E,S2)) == conc(app(E,empty),linear(S))

==(linear(empty()),empty())
if is_list(E) then
    ==(linear(app(E,S2)),conc(linear(E),linear(S)))
if not(is_list(E)) then
    ==(linear(app(E,S2)),conc(app(E,empty()),linear(S)))

==(linear(empty()),empty())
==(linear(app(E,S2)),conc(linear(E),linear(S)))
                    <= is_list(E)
==(linear(app(E,S2)),conc(app(E,empty()),linear(S)))
                    <= not(is_list(E))
```

```
eval(linear(empty()),R) <=> R=empty()
eval(linear(app(E,S2)),R) <= is_list(E) &
                        eval(conc(linear(E),linear(S)),R)
eval(linear(app(E,S2)),R) <= not(is_list(E)) &
                        eval(conc(app(E,empty()),linear(S)),R)

eval(linear(S),R) <= eval(S,S1) & S1=empty()  & R=empty()
eval(linear(S),R) <= eval(S,S1) & S1=app(E,S2) & is_list(E)
                        & eval(conc(linear(E),linear(S)),R)
eval(linear(S),R) <= eval(S,S1) & S1=app(E,S2) &
                        not(is_list(E)) &
                        eval(conc(app(E,empty()),linear(S)),R)

eval(linear(S),[]) :- eval(S,[]).
eval(linear(S),R) :- eval(S,[E|S2]),is_list(E),
                        eval(conc(linear(E),linear(S)),R).
eval(linear(S),R) :- eval(S,[E|S2]),not(is_list(E)),
                        eval(conc([E],linear(S)),R).

linear([],[]).
linear([E|S2],R) :- is_list(E),linear(E,E1),
                    linear([E|S2],S1),conc(E1,S1,R).
linear([E|S2],R) :- not(is_list(E)),linear([E|S2],S1),
                    conc([E],S1,R).
```

fonction sort(S,P) donne Sr .

Avec P : PRED,
S,Sr : SEQ,

Cette fonction donne une liste Sr qui est la liste S dont tous les élements sont ordonnés selon une propriété P.

Préconditions : N > 0, N <= length(L)

Paramètre_d'induction S

Relation_bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

S cons1 : empty()
cons2 : app(E,S2)

Construction_des_Fi

- si S est empty(), Sr est empty(), puisque la liste vide n'a pas a être triée.
- si S est app(E,S2), on fait appel à une autre fonction que l'on considère existante et correcte : insert (voir plus bas). On tri d'abord S2, puis on insère l'élément E dans la liste triée, au bon endroit selon P.

Axiomes_obtenus

```
sort(empty(),P)    == empty()
sort(app(E,S2),P) == insert(E,sort(S2,P),P)

==(sort(empty(),P),empty())
==(sort(app(E,S2),P),insert(E,sort(S2,P),P))

eval(sort(empty(),P),R) <=> R=empty()
eval(sort(app(E,S2),P),R) <=> eval(insert(E,sort(S2,P),P),R)

eval(sort(S,P),R) <= eval(S,S1) & S1=empty() & R=empty()
eval(sort(S,P),R) <= eval(S,S1) & S1=app(E,S2) &
                     eval(insert(E,sort(S2,P),P),R)

eval(sort(S,P),[]) :- eval(S,[]).
eval(sort(S,P),R) :- eval(S,[E|S2]),
                     eval(insert(E,sort(S2,P),P),R).

sort([],_,[]).
sort([E|S2],P,R) :- sort(S2,P,R1),insert(E,R1,R).
```

fonction filter(S,P) donne Sr .

Avec P : PRED,
S,Sr : SEQ,

Cette fonction donne une liste Sr qui est la sous-liste des éléments de la liste S, qui satisfont tous la même propriété P.

Paramètre_d'induction S

Relation_bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

```
     S    cons1 : empty()
          cons2 : app(E,S2)
```

Construction_des_Fi

- si S est empty(), Sr est aussi empty().
- si S est app(E,S2), on filtre la liste queue et si E satisfait à la propriété, l'élément de tête est ajouté à Sr.

Axiomes_obtenus

```
                    filter(empty(),P) == empty()
if p(E)       then filter(app(E,S2),P) == app(E,filter(S2,P))
if not(p(E)) then filter(app(E,S2),P) == filter(S2,P)

                    filter(empty(),P) == empty()
if p(E)       then filter(app(E,S2),P) == app(E,filter(S2,P))
if not(p(E)) then filter(app(E,S2),P) == filter(S2,P)

==(filter(empty(),P),empty())
==(filter(app(E,S2),P),app(E,filter(S2,P))) <= p(E)
==(filter(app(E,S2),P),filter(S2,P)) <= not(p(E))

eval(filter(empty(),P),R) <=> R=empty()
eval(filter(app(E,S2),P),R) <= p(E) &
                               eval(app(E,filter(S2,P)),R)
eval(filter(app(E,S2),P),R) <= not(p(E)) &
                               eval(filter(S2,P),R)

eval(filter(S,P),R) <= eval(S,S1) & S1=empty() & R=empty()
eval(filter(S,P),R) <= eval(S,S1) & S1=app(E,S2) & p(E) &
                               eval(app(E,filter(S2,P)),R)
eval(filter(S,P),R) <= eval(S,S1) & S1=app(E,S2) & not(p(E))
                       & eval(filter(S2,P),R)

eval(filter(S,P),[])     :- eval(S,[]).
eval(filter(S,P),[E|I]) :- eval(S,[E|S2]),p(E),
                           eval(filter(S2,P),I).
eval(filter(S,P),R)      :- eval(S,[E|S2]),not(p(E)),
                           eval(filter(S2,P),R).

filter([],_,[]).
filter([E|S2],P,[E|I]) :- p(E),filter(S2,P,I).
filter([E|S2],P,R) :- not(p(E)),filter(S2,P,R).
```

fonction merge(S,S2,P) donne Sr.

Avec P : PRED,
S,S2,Sr : SEQ,

Cette fonction donne une liste SR qui est la fusion des listes S et S2 avec un ordre déterminé par une propriété P. Les liste S et S2 sont supposées triées selon cette propriété.

Paramètres d'induction S et S2

Relation bien-fondée

(s1,s2) < (s1*,s2*) ssi s2 est un suffixe propre de s2* ou s2=s2* et s1 est un propre suffixe de s1*.
L'élément minimal est (empty(),empty()).

Forme structurelle des paramètres d'induction

S	cons1	:	empty()
	cons2	:	app(E,S3)
S2	cons1	:	empty()
	cons2	:	app(E2,S4)

Construction des Fi

- si S est empty(), la liste résultante est S2.
- si S2 est empty(), la liste résultante est S.
- si S est app(E,S3) et S2 est app(E2,S4), Il faut tester si la propriété de l'élément en tête de la liste S est plus petit que celle du sommet de la liste S2.

P(E)<=P(E2) et Sr=app(E2,merge(S,S4,P))
P(E)>P(E2) et Sr=app(E,merge(S3,S2,P))

Remarque : Respect de la relation bien-fondée.
L'emploi de merge(S,S4,SR) est permis avec (S,S4)<(S,S2).
L'emploi de merge(S3,S2,SR) est permis avec (S3,S2)<(S,S2).

Axiomes obtenus

merge(empty(),S2,P) == S2
merge(S,empty(),P) == S
if P(E)<=P(E2) then
merge(app(E,S3),app(E2,S4))) == app(E2,merge(S,S4,P))
if P(E)>P(E2) then
merge(app(E,S3),app(E2,S4))) == app(E,merge(S3,S2,P))

==(merge(empty(),S2,P),S2)
==(merge(S,empty(),P),S)
if p(E)<=p(E2) then
==(merge(app(E,S3),app(E2,S4),P),app(E2,merge(S,S4,P)))
if p(E)>p(E2) then
==(merge(app(E,S3),app(E2,S4),P),app(E,merge(S3,S2,P)))

```
==(merge(empty(),S2,P),S2)
==(merge(S,empty(),P),S)
==(merge(app(E,S3),app(E2,S4),P),app(E2,merge(S,S4,P))) <=
              p(E)<=p(E2)
==(merge(app(E,S3),app(E2,S4),P),app(E,merge(S3,S2,P))) <=
              p(E)>p(E2)

eval(merge(empty(),S2,P),R) <=> eval(S2,R)
eval(merge(S,empty(),P),R) <=> eval(S,R)
eval(merge(app(E,S3),app(E2,S4),P),R) <= p(E)<=p(E2) &
                              eval(app(E2,merge(S,S4,P)),R)
eval(merge(app(E,S3),app(E2,S4),P),R) <= p(E)>p(E2)  &
                              eval(app(E,merge(S3,S2,P)),R)

eval(merge(S,S2,P),R) <= eval(S,S1) & S1=empty() &
                         eval(S2,R)
eval(merge(S,S2,P),R) <= eval(S2,S3) & S3=empty() &
                         eval(S,R)
eval(merge(S,S2,P),R) <= eval(S,S1) & S1=app(E,S3) &
                         eval(S2,S5) & S5=app(E2,S4) &
                         p(E)<=p(E2) &
                         eval(app(E2,merge(S,S4,P)),R)
eval(merge(S,S2,P),R) <= eval(S,S1) & S1=app(E,S3) &
                         eval(S2,S5) & S5=app(E2,S4) &
                         p(E)>p(E2)  &
                         eval(app(E,merge(S3,S2,P)),R)

eval(merge(S,S2,P),R) :- eval(S,[]),eval(S2,R).
eval(merge(S,S2,P),R) :- eval(S2,[]),eval(S,R).
eval(merge(S,S2,P),[E2|I]) :- eval(S,[E|S3]),
                              eval(S2,[E2|S4]),p(E)<=p(E2),
                              eval(merge(S,S4,P),I).
eval(merge(S,S2,P),[E|I])  :- eval(S,[E|S3]),
                              eval(S2,[E2|S4]),p(E)>p(E2),
                              eval(merge(S3,S2,P),I).

merge([],S2,_,S2).
merge(S,[],_,S).
merge([E|S],[E2|S4],P,[E2|I]) :- p(E)<=p(E2),
                                 merge([E|S],S4,P,I).
merge([E|S3],[E2|S4],P,[E|I]) :- p(E)>p(E2),
                                 merge(S3,[E2|S4],P,I).
```

fonction ith(N,S) donne Er .

Avec N : NAT,
S : SEQ,
Er : ELEM.

Cette fonction donne un élément E qui est à la position N dans la liste S.

Préconditions : N > zero(), N <= leng(S),
is_empty(S) == false()

Paramètre_d'induction N

Relation_bien-fondée

n1<n2 ssi n1 a une valeur plus petite que n2
(De par la précondition, l'élément minimal est succ(zero()), et pas zero())

Forme_structurelle_du_paramètre_d'induction

N cons1 : succ(zero())
cons2 : succ(...(succ(zero()))...)

Construction_des_Fi

- si N est succ(zero()), alors, on fournit l'élément E en tête de la liste
- si N est succ(...(succ(zero()))...),
on doit chercher l'élément dans la liste suite, mais à la position précédente de N.

Axiomes_obtenus

```
ith(succ(zero()),app(E,S2)) == E
ith(N,app(E,S2))             == ith(pred(N),S2)

==(ith(succ(zero()),app(E,S2)),E)
==(ith(N,app(E,S2)),ith(pred(N),S2))

eval(ith(succ(zero()),app(E,S2)),R) <=> eval(E,R)
eval(ith(N,app(E,S2)),R) <=> eval(ith(pred(N),S2),R)

eval(ith(N,S),R) <= eval(N,N1) & N1=succ(zero()) &
                    eval(S,S1) & S1=app(E,S2) & eval(E,R)
eval(ith(N,S),R) <= eval(S,S1) & S1=app(E,S2) &
                    eval(ith(pred(N),S2),R)

eval(ith(N,S),R) :- eval(N,1),eval(S,[E|S2]),eval(E,R).
eval(ith(N,S),R) :- eval(S,[E|S2]),
                    eval(ith(pred(N),S2),R).

ith(1,[E|_],E).
ith(N,[_|S2],R) :- N2=N-1,ith(N2,S2,R).
```

```
fonction head(S) donne Er .

Avec L    : SEQ,
     Er   : ELEM.

Cette fonction donne un élément Er qui est en
tête de la liste S.

Préconditions :  is_empty(S) == false()
```

Paramètre_d'induction S

Relation_bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

Il ne faut pas envisager le cas empty() selon la pré-condition.

S cons1 : app(E,S2)

Construction_des_Fi

- si S est app(E,S2), alors E est l'élément de la tête

Axiomes_obtenus

```
head(app(E,S2)) == E

==(head(app(E,S2)),E)

eval(head(app(E,S2)),R) <=> eval(E,R)

eval(head(S),R) <= eval(S,S1) & S1=app(E,S2) & eval(E,R)

eval(head(S),R) :- eval(S,[E|S2]),eval(E,R).

eval(head(S),R) :- eval(S,S1),head(S1,R).

head([E|_],E).
```

fonction last(S) donne Er .

Avec S : SEQ,
Er : ELEM.

Cette fonction donne un élément E qui est en queue de la liste S.

Préconditions : is_empty(S) = false()

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

Il ne faut pas envisager le cas empty() selon la précondition.

S cons1 : app(E,S2)

Construction_des_Fi

- si S est app(E,S2), si S2 est la liste vide, alors E est l'élément de la tête, sinon, il faut rechercher la tête dans la liste queue.

Axiomes_obtenus

```
last(app(E,empty())) == E
last(app(E,S2))      == last(S2)

==(last(app(E,empty())),E)
==(last(app(E,S2)),last(S2))

eval(last(app(E,empty())),R) ‹=› eval(E,R)
eval(last(app(E,S2)),R) ‹=› eval(last(S2),R)

eval(last(S),R) ‹= eval(S,S1) & S1=app(E,empty()) &
                   eval(E,R)
eval(last(S),R) ‹= eval(S,S1) & S1=app(E,S2)  &
                   eval(last(S2),R)

eval(last(S),R) :- eval(S,[E]),eval(E,R).
eval(last(S),R) :- eval(S,[E¦S2]),eval(last(S2),R).

last([E],E).
last([_¦S2],R) :- last(S2,R).
```

fonction length(L) donne Nr .

Avec Nr : NAT,
L : SEQ,

Cette fonction donne un entier naturel Nr qui est le nombre d'éléments de la liste L (ou longueur).

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

S cons1 : empty()
cons2 : app(E,S2)

Construction_des_Fi

- si S est empty(), NR est zero().
- si S est app(E,S2), on prend le successeur de la longeur de la liste queue S2.

Axiomes_obtenus

```
length(empty())   == zero()
length(app(E,S2)) == succ(length(S2))

==(length(empty()),zero())
==(length(app(E,S2)),succ(length(S2)))

eval(length(empty()),R) <=> R=zero()
eval(length(app(E,S2)),R) <=> eval(succ(length(S2)),R)

eval(length(S),R) <= S=empty() & R=zero()
eval(length(S),R) <= S=app(E,S2) & eval(succ(length(S2)),R)

eval(length(S),R) <= eval(S,S1) & S1=empty() & R=zero()
eval(length(S),R) <= eval(S,S1) & S=app(E,S2) &
                     eval(succ(length(S2)),R)

eval(length(S),R) :- eval(S,S1),S1=[],R=0.
eval(length(S),R) :- eval(S,S1),S=[E|S2],
                     eval(succ(length(S2)),R).

eval(length(S),0) :- eval(S,[]).
eval(length(S),R) :- eval(S,[E|S2]),eval(length(S2),L),
                     R is L+1.

length([],0).
length([E|S2],R) :- length(S2,R2),R=R2+1.
```

fonction is_in(E,S) donne Br .

Avec Br : BOOL,
S : SEQ,
E : ELEM.

Cette fonction donne une valeur booléenne qui est vrai si l'élement E est dans la liste S, faux sinon

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

S cons1 : empty()
cons2 : app(E,S2)

Construction_des_Fi

- si S est empty(), Br est false(),
car aucun élément n'appartient à la liste vide.
- si S est app(E2,S2), Deux cas se présentent :
si E=E2, alors BR est true() sinon il faut vérifier si l'élément n'est pas dans la liste queue S2

Axiomes_obtenus

```
is_in(E,empty())     == false()
is_in(E,app(E,S2))   == true()
is_in(E,app(E2,S2)) == is_in(E,S2)

==(is_in(E,empty()),false())
==(is_in(E,app(E,S2)),true())
==(is_in(E,app(E2,S2)),is_in(E,S2))

eval(is_in(E,empty()),R) <=> R=false()
eval(is_in(E,app(E,S2)),R) <=> R=true()
eval(is_in(E,app(E2,S2)),R) <=> eval(is_in(E,S2),R)

eval(is_in(E,S),R) <= eval(S,S1) & S1=empty() & R=false()
eval(is_in(E,S),R) <= eval(S,S1) & S1=app(E,S2) & R=true()
eval(is_in(E,S),R) <= eval(S,S1) & S1=app(E2,S2) &
                      eval(is_in(E,S2),R)
```

```
eval(is_in(E,S),R) <= eval(S,[]),R=false().
eval(is_in(E,S),R) <= eval(S,[E¦S2]),R=true().
eval(is_in(E,S),R) <= eval(S,[E2¦S2]),eval(is_in(E,S2),R).

eval(is_in(E,S),false) :- eval(S,[]).
eval(is_in(E,S),true) :- eval(S,[E¦S2]).
eval(is_in(E,S),R) :- eval(S,[E2¦S2]),eval(is_in(E,S2),R).

is_in(_,[],false).
is_in(E,[E¦_],true).
is_in(E,[_¦S2],R) :- is_in(E,S2,R).
```

```
fonction is_empty(S) donne Br.

Avec Br   : BOOL,
     S    : SEQ,

Cette fonction donne une valeur booléenne qui est
vrai si la liste S est vide, faux sinon
```

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

```
S    cons1 :  empty()
     cons2 :  app(E,S2)
```

Construction_des_Fi

- si S est empty(), Br est vrai.
- si S est app(E,S2), Br est faux.

Axiomes_obtenus

```
is_empty(empty())   == true()
is_empty(app(E,S2)) == false()

==(is_empty(empty()),true())
==(is_empty(app(E,S2)),false())

eval(is_empty(empty()),R) <=› R=true()
eval(is_empty(app(E,S2)),R) <=› R=false()
```

```
eval(is_empty(S),R) <= eval(S,S1) & S1=empty() & R=true()
eval(is_empty(S),R) <= eval(S,S1) & S1=app(E,S2) & R=false()

eval(is_empty(S),true) :- eval(S,[]).
eval(is_empty(S),false) :- eval(S,[E|S2]).

is_empty([],true).
is_empty([_|_],false).
```

fonction is_sort(S,P) donne Br.

Avec Br : BOOL,
S : SEQ,

Cette fonction donne une valeur booléenne qui est vrai si la liste S est triée selon le prédicat P, faux sinon

Paramètre d'induction S

Relation bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

S cons1 : empty()
cons2 : app(E,S2)

Construction des Fi

- si S est empty(), Br est vrai.
- si S est app(E,S2), il faut d'abord voir si la queue possède encore des éléments.

Si oui, il faut comparer E avec l'élément suivant.
Si il est plus petit selon le prédicat, on vérifie que la queue est également triée.
Sinon, la liste n'est pas triée.
Si non, une liste de 1 élément est toujours triée.

Axiomes_obtenus

```
is_sort(empty()) == true()
is_sort(app(E,empty())) == true()
if p(E)<=p(E2) then
     is_sort(app(E,app(E2,S2)) == is_sort(app(E2,S2))
if p(E)>p(E2) then
     is_sort(app(E,app(E2,S2)) == false()

==(is_sort(empty(),P),true())
==(is_sort(app(E,empty()),P),true())
if p(E)<=p(E2) then
     ==(is_sort(app(E,app(E2,S2),P),is_sort(app(E2,S2),P))
if p(E)>p(E2) then
     ==(is_sort(app(E,app(E2,S2),P),false())

==(is_sort(empty(),P),true())
==(is_sort(app(E,empty()),P),true())
==(is_sort(app(E,app(E2,S2),P),is_sort(app(E2,S2),P)) <=
                p(E)<=p(E2)
==(is_sort(app(E,app(E2,S2),P),false()) <= p(E)>p(E2)

eval(is_sort(empty(),P),R) <=> R=true()
eval(is_sort(app(E,empty()),P),R) <=> R=true()
eval(is_sort(app(E,app(E2,S2),P),R) <=
               p(E)<=p(E2) & eval(is_sort(app(E2,S2),P),R)
eval(is_sort(app(E,app(E2,S2),P),R) <= p(E)>p(E2) &
               R=false()

eval(is_sort(S,P),R) <= eval(S,S1) & S1=empty() & R=true()
eval(is_sort(S,P),R) <= eval(S,S1) & S1=app(E,empty()) &
                        R=true()
eval(is_sort(S,P),R) <= eval(S,S1) & S1=app(E,app(E2,S2)) &
                  p(E)<=p(E2) & eval(is_sort(app(E2,S2),P),R)
eval(is_sort(S,P),R) <=  eval(S,S1) & S1=app(E,app(E2,S2)) &
                           p(E)>p(E2) & R=false()

eval(is_sort(S,P),true) :- eval(S,[]).
eval(is_sort(S,P),true) :- eval(S,[E]).
eval(is_sort(S,P),R) :- eval(S,[E|[E2|S2]]),p(E)<=p(E2),
                        eval(is_sort([E2|S2],P),R).
eval(is_sort(S,P),false) :- eval(S,[E|[E2|S2]]),p(E)>p(E2).

is_sort([],_,true).
is_sort([E],_,true).
is_sort([E|[E2|S2]],P,R) :- p(E)<=p(E2),
                            is_sort([E2|S2],P,R).
is_sort([E|[E2|S2]],P,false) :- p(E)>p(E2).
```

fonction insert(E,S,P) donne Sr.

privée au module liste

Avec P : PRED,
S,Sr : SEQ,
E : ELEM.

Cette fonction donne une liste Sr qui est la liste S déjà triée à laquelle on insère l'élément E au bon endroit de L selon une propriété P.

Préconditions : is_sort(S,P) == true()

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

S cons1 : empty()
cons2 : app(E,S2)

Construction_des_Fi

- si S est empty(), On ajoute l'élément à la liste, puisqu'il n'y a aucun autre élément de comparaison.
- si S est app(E2,S2), On compare les deux éléments E et E2 selon la propriété P. Deux cas se présentent :
p(E)‹=p(E2), alors on ajoute l'élément E2 à la liste triée S2 où E à été inséré.
p(E)›p(E2), alors on ajoute l'élément E à la liste triée S2 où E2 à été inséré.

Axiomes_obtenus

```
insert(E,empty(),P) == app(E,empty())
if p(E)‹=p(E2) then
    insert(E,app(E2,S2),P) == app(E2,insert(E,S2,P))
if p(E)›p(E2) then
    insert(E,app(E2,S2),P) == app(E,insert(E2,S2,P))

==(insert(E,empty(),P),app(E,empty()))
if p(E)‹=p(E2) then
    ==(insert(E,app(E2,S2),P),app(E2,insert(E,S2,P)))
if p(E)›p(E2) then
    ==(insert(E,app(E2,S2),P),app(E,insert(E2,S2,P)))
```

```
==(insert(E,empty(),P),app(E,empty()))
==(insert(E,app(E2,S2),P),app(E2,insert(E,S2,P))) <=
                  p(E)<=p(E2)
==(insert(E,app(E2,S2),P),app(E,insert(E2,S2,P))) <=
                  p(E)>p(E2)

eval(insert(E,empty(),P),R) <=> eval(app(E,empty()),R)
eval(insert(E,app(E2,S2),P),R) <= p(E)<=p(E2) &
                        eval(app(E2,insert(E,S2,P)),R)
eval(insert(E,app(E2,S2),P),R) <= p(E)>p(E2) &
                        eval(app(E,insert(E2,S2,P)),R)

eval(insert(E,S,P),R) <= eval(S,S1) & S1=empty()  &
                         eval(app(E,empty()),R)
eval(insert(E,S,P),R) <= eval(S,S1) & S1=app(E2,S2) &
                         p(E)<=p(E2) &
                         eval(app(E2,insert(E,S2,P)),R)
eval(insert(E,S,P),R) <= eval(S,S1) & S1=app(E2,S2) &
                         p(E)>p(E2) &
                         eval(app(E,insert(E2,S2,P)),R)

eval(insert(E,S,P),[E]) :- eval(S,[]).
eval(insert(E,S,P),[E2|I]) :- eval(S,[E2|S2]),p(E)<=p(E2),
                              eval(insert(E,S2,P),I).
eval(insert(E,S,P),[E|I]) :- eval(S,[E2|S2]),p(E)>p(E2),
                             eval(insert(E2,S2,P),I).

insert(E,[],_,[E]).
insert(E,[E2|S2],P,[E2|I]) :- p(E)<=p(E2), insert(E,S2,P,I).
insert(E,[E2|S2],P,[E|I])  :- p(E)>p(E2), insert(E2,S2,P,I).
```

fonction correct(E,N,S) donne Sr.

Avec S,Sr : SEQ.
E : ELEM.
N : NAT.

Cette fonction donne une liste Sr qui est la liste des éléments de la liste S, excepté pour l'élément à la position N qui est remplacé par l'élément E.

préconditions : N >= succ(zero), N <= length(S).

Paramètre_d'induction N

Relation_bien-fondée

n1<n2 si n1 a une valeur plus petite que n2

Forme_structurelle_du_paramètre_d'induction

N cons1 : succ(zero())
cons2 : succ(...(succ(zero()))...)

Construction_des_Fi

- si N=succ(zero()), Sr est la concaténation de E avec la queue de la liste.
- si N=succ(N1), j'ajoute à Sr l'élément en tête de liste et j'applique la fonction correct sur la queue de S avec la position N1.

axiomes_obtenus

```
correct(E,zero,S)    ==  ERREUR "position invalide, hors des
                         limites"
correct(E,succ(zero()),empty())== ERREUR "position invalide,
                                  hors des limites"
correct(E,succ(zero()),app(E2,S2)) == app(E,S2)
correct(E,succ(N1),app(E2,S2)) == app(E2,correct(E,N1,S2))

==(correct(E,succ(zero()),app(E2,S2)),app(E,S2))
==(correct(E,succ(N1),app(E2,S2)),app(E2,correct(E,N1,S2)))

eval(correct(E,succ(zero()),app(E2,S2)),Sr) <=> Sr=app(E,S2)
eval(correct(E,succ(N1),app(E2,S2)),Sr) <=>
                          eval(app(E2,correct(E,N1,S2)),Sr)

eval(correct(E,N,S),Sr) <= eval(N,succ(zero)) &
                           eval(S,app(E2,S2)) & Sr=app(E,S2)
eval(correct(E,N,S),Sr) <= eval(N,succ(N1)) &
                           eval(S,app(E2,S2)) &
                           eval(app(E2,correct(E,N1,S2)),Sr)
```

```
eval(correct(E,N,S),[E|S2]) :- eval(N,succ(zero())),
                               eval(S,[E2|S2]).
eval(correct(E,N,S),[E2|Sr]) :- eval(N,succ(N1)),
                                eval(S,[E2|S2]),
                                eval(correct(E,N1,S2),Sr).

eval(correct(E,N,S),[E|S2]) :- eval(N,N1), N1 is 1,
                               eval(S,[E2|S2]).
eval(correct(E,N,S),[E2|Sr]) :- eval(N,N2), N1 is N2-1,
                                eval(S,[E2|S2]),
                                eval(correct(E,N1,S2),Sr).

correct(E,1,[_|S2],[E|S2]).
correct(E,N,[E2|S2],[E2|Sr]) :- N1 is N-1, correct(E,N1,S2,Sr).
```

fonction subst_all(E,E2,S) donne Sr.

Avec S,Sr : SEQ.
E,E2 : ELEM.

Cette fonction donne une liste Sr qui est la liste S dans laquelle toutes les occurences de l'élément E ont été remplacées par E2.

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

S cons1 : empty()
cons2 : app(E,S2)
cons3 : app(E3,S2)

Construction_des_Fi

- si S=empty(), alors il n'y a rien a substituer. La liste résultante est vide.
- si S=app(E,S2), je dois substituer l'élément E par E2, puis substituer la suite S2 de la liste
- si S=app(E3,S2), je conserve l'élément E3 et je substitue les occurences de E par E2 dans la suite S2.

axiomes_obtenus

```
subst_all(E,E2,empty()) == empty()
subst_all(E,E2,app(E,S2)) == app(E2,subst_all(E,E2,S2))
subst_all(E,E2,app(E3,S2)) == app(E3,subst_all(E,E2,S2))
```

```
==(subst_all(E,E2,empty()),empty())
==(subst_all(E,E2,app(E,S2)),app(E2,subst_all(E,E2,S2)))
==(subst_all(E,E2,app(E3,S2)),app(E3,subst_all(E,E2,S2)))

eval(subst_all(E,E2,empty()),Sr) <=> Sr=empty()
eval(subst_all(E,E2,app(E,S2)),Sr) <=>
                          eval(app(E2,subst_all(E,E2,S2)),Sr)
eval(subst_all(E,E2,app(E3,S2)),Sr) <=>
                          eval(app(E3,subst_all(E,E2,S2)),Sr)

eval(subst_all(E,E2,S),Sr) <= eval(S,empty()) & Sr=empty()
eval(subst_all(E,E2,S),Sr) <= eval(S,app(E,S2)) &
                          eval(app(E2,subst_all(E,E2,S2)),Sr)
eval(subst_all(E,E2,S),Sr) <= eval(S,app(E3,S2)) &
                          eval(app(E3,subst_all(E,E2,S2)),Sr)

eval(subst_all(E,E2,S),[]) :- eval(S,[]).
eval(subst_all(E,E2,S),[E2¦Sr]) :- eval(S,[E¦S2]),
                                   eval(subst_all(E,E2,S2),Sr).
eval(subst_all(E,E2,S),[E3¦Sr]) :- eval(S,[E3¦S2]),
                                   eval(subst_all(E,E2,S2),Sr).

subst_all(E,E2,[],[]).
subst_all(E,E2,[E¦S2],[E2¦Sr]) :- subst_all(E,E2,S2,Sr).
subst_all(E,E2,[E3¦S2],[E3¦Sr]) :- subst_all(E,E2,S2,Sr).
```

fonction subst_one(E,E2,S) donne Sr.

Avec E,E2 : ELEM,
     S,Sr : SEQ,

Cette fonction donne une liste Sr qui est la liste S dans laquelle la première occurence de l'élément E a été remplacée par E2.

Paramètre_d'induction S

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

```
S      cons1 :  empty()
       cons2 :  app(E,S2)
       cons3 :  app(E3,S2)
```

Construction_des_Fi

- si S=empty(), alors il n'y a rien a substituer. La liste résultante est vide.
- si S=app(E,S2), je dois substituer l'élément E par E2, puis arrêter, puisque j'ai une occurence de E.
- si S=app(E3,S2), je conserve l'élément E3 et je substitue les occurences de E par E2 dans la suite S2.

Axiomes_obtenus

```
subst_one(E,E2,empty()) == empty()
subst_one(E,E2,app(E,S2)) == app(E2,S2)
subst_one(E,E2,app(E3,S2)) == app(E3,subst_one(E,E2,S2))

==(subst_one(E,E2,empty()),empty())
==(subst_one(E,E2,app(E,S2)),app(E2,S2))
==(subst_one(E,E2,app(E3,S2)),app(E3,subst_one(E,E2,S2)))

eval(subst_one(E,E2,empty()),Sr) <=> Sr=empty()
eval(subst_one(E,E2,app(E,S2)),Sr) <=> Sr = app(E2,S2)
eval(subst_one(E,E2,app(E3,S2)),Sr) <=>
                          eval(app(E3,subst_one(E,E2,S2)),Sr)

eval(subst_one(E,E2,S),Sr) <= eval(S,empty()) & Sr=empty()
eval(subst_one(E,E2,S),Sr) <= eval(S,app(E,S2)) &
                              Sr = app(E2,S2)
eval(subst_one(E,E2,S),Sr) <= eval(S,app(E3,S2)) &
                          eval(app(E3,subst_one(E,E2,S2)),Sr)

eval(subst_one(E,E2,S),[]) :- eval(S,[]).
eval(subst_one(E,E2,S),[E2|S2]) :- eval(S,[E|S2]).
eval(subst_one(E,E2,S),[E3|Sr]) :- eval(S,[E3|S2]),
                                   eval(subst_one(E,E2,S2),Sr).

subst_one(E,E2,[],[]).
subst_one(E,E2,[E|S2],[E2|S2]).
subst_one(E,E2,[E3|S2],[E3|Sr]) :- subst_one(E,E2,S2,Sr).
```

fonction subcut(E,E2,S) donne Sr.

Avec S,Sr : SEQ,
E,E2 : ELEM.

Cette fonction donne une liste Sr qui est la sous-liste éléments de S compris entre les premières occurences des éléments E et E2.

Paramètre_d'induction S

Relation_bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

```
S      cons1 :  empty()
       cons2 :  app(E,S2)
       cons3 :  app(E3,S2)
```

Construction_des_Fi

- si S=empty(), alors il n'y a rien a conserver. La liste résultante est vide.
- si S=app(E,S2), je dois conserver l'élément E, puis conserver la suite S2 depuis le premier élément de S2 jusqu'à la première occurence de E2 (fonction prefcut)
- si S=app(E3,S2), je cherche la première occurence de E (je ne suis pas encore dans la zone de E à E2).

axiomes_obtenus

```
subcut(E,E2,empty()) == empty()
subcut(E,E2,app(E,S2)) == app(E,prefcut(E2,S2))
subcut(E,E2,app(E3,S2)) == subcut(E,E2,S2)

==(subcut(E,E2,empty()),empty())
==(subcut(E,E2,app(E,S2)),app(E,prefcut(E2,S2)))
==(subcut(E,E2,app(E3,S2)),subcut(E,E2,S2))

eval(subcut(E,E2,empty()),Sr) <=> Sr=empty()
eval(subcut(E,E2,app(E,S2)),Sr)<=>eval(app(E,prefcut(E2,S2)),Sr)
eval(subcut(E,E2,app(E3,S2)),Sr) <=> eval(subcut(E,E2,S2),Sr)

eval(subcut(E,E2,S),Sr) <= eval(S,empty()) & Sr=empty()
eval(subcut(E,E2,S),Sr) <= eval(S,app(E,S2)) &
                           eval(app(E,prefcut(E2,S2)),Sr)
eval(subcut(E,E2,S),Sr) <= eval(S,app(E3,S2)) &
                           eval(subcut(E,E2,S2),Sr)

eval(subcut(E,E2,S),[]) :- eval(S,[]).
eval(subcut(E,E2,S),[E|Sr]) :- eval(S,[E|S2]),
                               eval(prefcut(E2,S2),Sr).
eval(subcut(E,E2,S),Sr) :- eval(S,[E3|S2]),
                           eval(subcut(E,E2,S2),Sr).

subcut(E,E2,[],[]).
subcut(E,E2,[E|S2],[E|Sr]) :- prefcut(E2,S2,Sr).
subcut(E,E2,[E3|S2],Sr) :- subcut(E,E2,S2,Sr).
```

fonction prefcut(E,S) donne Sr.

Avec S,Sr : SEQ,
E,E2 : ELEM.

Cette fonction donne une liste Sr qui est la sous-liste éléments de S jusqu'à la première occurence de l'élément E.

Paramètre_d'induction S

Relation_bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

S cons1 : empty()
cons2 : app(E,S2)
cons3 : app(E3,S2)

Construction_des_Fi

- si S=empty(), alors il n'y a rien a conserver. La liste résultante est vide.
- si S=app(E,S2), je dois conserver l'élément E, qui est la fin de la sous-liste d'intérêt. Donc on peut arrêter ici.
- si S=app(E3,S2), je conserve l'élément E3 et je recommence l'opération jusqu'à la rencvontre de l'élément E.

axiomes_obtenus

```
prefcut(E,empty()) == empty()
prefcut(E,app(E,S2)) == app(E,empty())
prefcut(E,app(E2,S2)) == app(E2,prefcut(E,S2))

==(prefcut(E,empty()),empty())
==(prefcut(E,app(E,S2)),app(E,empty()))
==(prefcut(E,app(E2,S2)),app(E2,prefcut(E,S2)))

eval(prefcut(E,empty()),Sr) <=> Sr=empty()
eval(prefcut(E,app(E,S2)),Sr) <=> Sr=app(E,empty())
eval(prefcut(E,app(E2,S2)),Sr) <=>
                          eval(app(E2,prefcut(E,S2)),Sr)

eval(prefcut(E,S),Sr) <= eval(S,empty()) & Sr=empty()
eval(prefcut(E,S),Sr) <= eval(S,app(E,S2))  & Sr=app(E,empty())
eval(prefcut(E,S),Sr) <= eval(S,app(E2,S2)) &
                          eval(app(E2,prefcut(E,S2)),Sr)
```

```
eval(prefcut(E,S),[]) :- eval(S,[]).
eval(prefcut(E,S),[E]) :- eval(S,[E|S2]).
eval(prefcut(E,S),[E2|Sr]) :- eval(S,[E2|S2]),
                              eval(prefcut(E,S2),Sr).

prefcut(E,[],[]).
prefcut(E,[E|S2],[E]).
prefcut(E,[E2|S2],[E2|Sr]) :- prefcut(E,S2,Sr).
```

fonction newseq(E,N) donne Sr.

Avec Sr : SEQ,
E : ELEM,
N : NAT.

Cette fonction donne une liste Sr qui comprend N éléments E.

Paramètre_d'induction N

Relation_bien-fondée

n1<n2 si n1 a une valeur plus petite que n2

Forme_structurelle_du_paramètre_d'induction

N cons1 : succ(zero())
cons2 : succ(...(succ(zero()))...)

Construction_des_Fi

- si N est zero(), je crée une liste de 0 éléments, donc une liste vide.
- si N est succ(N1), j'ajoute un élément E au résultat de la création d'une suite de N1 élément E

axiomes_obtenus

```
newseq(E,zero()) == empty()
newseq(E,succ(N1)) == app(E,newseq(E,N1))

==(newseq(E,zero()),empty())
==(newseq(E,succ(N1)),app(E,newseq(E,N1)))

eval(newseq(E,zero()),Sr) <=> Sr=empty()
eval(newseq(E,succ(N1)),Sr) <=> eval(app(E,newseq(E,N1)),Sr)

eval(newseq(E,N),Sr) <= eval(N,zero()) & Sr=empty()
eval(newseq(E,N),Sr) <= eval(N,succ(N1))  &
                        eval(app(E,newseq(E,N1)),Sr)

eval(newseq(E,N),[]) :- eval(N,0).
eval(newseq(E,N),[E|Sr]) :- eval(N,N2), N1 is N2-1,
                            eval(newseq(E,N1),Sr).

newseq(E,0,[]).
newseq(E,N,[E|Sr]) :- N1 is N-1, newseq(E,N1,Sr).
```

fonction times(E,S) donne Nr.

Avec S : SEQ,
E : ELEM,
Nr : NAT.

Cette fonction donne un naturel Nr qui est le nombre d'occurence de l'élément E dans la liste S.

Paramètre d'induction S

Relation bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

S cons1 : empty()
cons2 : app(E,S2)
cons3 : app(E2,S2)

Construction des Fi

- si S est empty(), il n'y a aucun élément dans la liste.
- si S est app(E,S2). on a trouvé une fois (de plus) l'élément E. On ajoute donc 1 au résultat du nombre d'occurences dans S2.
- si S est app(E2,S2), le nombre d'occurences de E est le même dans app(E2,S2) que dans S2.

axiomes obtenus

```
times(E,empty()) == zero()
times(E,app(E,S2)) == succ(times(E,S2))
times(E,app(E2,S2)) == times(E,S2)

==(times(E,empty()),zero())
==(times(E,app(E,S2)),succ(times(E,S2)))
==(times(E,app(E2,S2)),times(E,S2))

eval(times(E,empty()),Nr) <=> Nr=zero()
eval(times(E,app(E,S2)),Nr) <=> eval(succ(times(E,S2)),Nr)
eval(times(E,app(E2,S2)),Nr) <=> eval(times(E,S2),Nr)

eval(times(E,S),Nr) <= eval(S,empty()) & Nr=zero()
eval(times(E,S),Nr) <= eval(S,app(E,S2)) &
                       eval(succ(times(E,S2)),Nr)
eval(times(E,S),Nr) <= eval(S,app(E2,S2)) &
                       eval(times(E,S2),Nr)
```

```
eval(times(E,S),0) :- eval(S,[]).
eval(times(E,S),Nr) :- eval(S,[E|S2]),
                       eval(times(E,S2),N1), Nr is N1 + 1.
eval(times(E,S),Nr) :- eval(S,[E2|S2]),
                       eval(times(E,S2),Nr).

times(E,[],0).
times(E,[E|S2],Nr) :- times(E,S2,N1), Nr is N1 + 1.
times(E,[E2|S2],Nr) :- times(E,S2,Nr).
```

fonction pos(E,S) donne Nr.

Avec S : SEQ,
E : ELEM,
Nr : NAT.

Cette fonction donne un naturel Nr qui est la position de l'élément E dans la liste S.

Paramètre d'induction S

Relation bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

S cons1 : empty()
cons2 : app(E,S2)
cons3 : app(E2,S2)

Construction des Fi

- si S est empty(), il n'y a aucun élément dans la liste.
- si S est app(E,S2). on a trouvé l'élément en tête de la liste. sa position est donc première.
- si S est app(E2,S2), on recherche la position de E dans la queue S et on ajoute 1 car on sait qu'il n'est pas en tête de la liste app(E2,S2)..

axiomes obtenus

```
pos(E,empty()) == zero()
pos(E,app(E,S2)) == succ(zero())
pos(E,app(E2,S2)) == succ(pos(E,S2))

==(pos(E,empty()),zero())
==(pos(E,app(E,S2)),succ(zero()))
==(pos(E,app(E2,S2)),succ(pos(E,S2)))

eval(pos(E,empty()),Nr) ‹=› Nr=zero()
eval(pos(E,app(E,S2)),Nr) ‹=› Nr = succ(zero())
eval(pos(E,app(E2,S2)),Nr) ‹=› eval(succ(pos(E,S2)),Nr)
```

```
eval(pos(E,S),Nr) <= eval(S,empty()) & Nr=zero()
eval(pos(E,S),Nr) <= eval(S,app(E,S2)) & Nr = succ(zero())
eval(pos(E,S),Nr) <= eval(S,app(E2,S2)) &
                     eval(succ(pos(E,S2)),Nr)

eval(pos(E,S),0) :- eval(S,[]).
eval(pos(E,S),1) :- eval(S,[E|S2]).
eval(pos(E,S),Nr) :- eval(S,[E2|S2]),
                     eval(pos(E,S2),N2), Nr is N2+1.

pos(E,[],0).
pos(E,[E|S2],1).
pos(E,[E2|S2],Nr) :- pos(E,S2,N2), Nr is N2+1.
```

fonction is_at(E,N,S) donne Br.

Avec S : SEQ,
E : ELEM,
N : NAT,
Br : BOOL.

Cette fonction donne un booléen Br qui est vrai si l'élément E est à la position N de la liste S, faux sinon.

Précondition : N >= succ(zero()), N <= length(S).

Paramètre_d'induction N,S

Relation_bien-fondée

n1<n2 si n1 a une valeur plus petite que n2

s1<s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

N	cons1	:	succ(zero())
	cons2	:	succ(...(succ(zero()))...)
S	cons1	:	empty()
	cons2	:	app(E,S2)

Construction_des_Fi

- si N est zero(), il y a une erreur dû à la précondition qui veut que l'opération a besoin comme argument d'une valeur comprise entre 1 et la longueur de S.
- si N2 est succ(zero()), alors le résultat est vrai si l'élément de tête de liste est celui que l'on a reçu en argument.
- si N est succ(...(succ(zero()))...) :
  - la liste n'est pas vide. Le résultat est identique au résultat de l'opération sur la queue de la séquence avec N diminué de 1 puisque l'on passe l'élément de tête.
  - la liste est vide. Le résultat est faux, puisqu'il n'y a pas d'élément dans la liste.

Axiomes_obtenus

```
is_at(E,zero,S) == ERREUR "valeur de paramètre fausse"
is_at(E,succ(zero()),app(E,S2)) == true()
is_at(E,succ(N1),empty()) == false()
is_at(E,succ(N1),app(E2,S2)) == is_at(E,N1,S2)
```

```
==(is_at(E,succ(zero()),app(E,S2)),true())
==(is_at(E,succ(N1),empty()),false())
==(is_at(E,succ(N1),app(E2,S2)),is_at(E,N1,S2))

eval(is_at(E,succ(zero()),app(E,S2)),Br) <=> Br=true()
eval(is_at(E,succ(N1),empty()),Br) <=> Br=false()
eval(is_at(E,succ(N1),app(E2,S2)),Br) <=>
                                  eval(is_at(E,N,S2),Br)

eval(is_at(E,N,S),Br) <= eval(N,succ(zero()) &
                         eval(S,app(E,S2)) & Br=true()
eval(is_at(E,N,S),Br) <= eval(N,succ(N1)) &
               eval(S,app(E2,S2)) & eval(is_at(E,N1,S2),Br)

eval(is_at(E,N,S),true) :- eval(N,1), eval(S,[E|S2]).
eval(is_at(E,N,S),Br) :- eval(N,N2), N1 is N2 -1,
                  eval(S,[E2|S2]), eval(is_at(E,N1,S2),Br).

is_at(E,1,[E|S2],true).
is_at(E,N,[E2|S2],Br) :- N1 is N -1, is_at(E,N1,S2,Br).
```

fonction is_sublist(S,S2) donne Br.

Avec S,S2 : SEQ,
     Br   : BOOL.

Cette fonction donne un booléen Br qui est vrai si la liste S est une sous-liste de mêmes éléments, dans le même ordre, de S2.

Paramètre d'induction S,S2

Relation bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

```
S     cons1 :  empty()
      cons2 :  app(E1,S1)

S2    cons1 :  empty()
      cons2 :  app(E3,S3)
```

Construction_des_Fi

- si S est vide et S2 aussi, alors S est une sous-séquence de S2.
- si S n'est pas vide mais que S2 l'est, alors S est une liste plus grande que S2, donc ce n'est pas une sous-séquence.
- si ni S ni S2 ne sont vides, et que leurs têtes sont les mêmes, il faut vérifier que la suite de S est un préfixe de la suite de S2.
- si ni S ni S2 ne sont vides, et que leurs têtes sont différentes, il faut vérifier que la suite de S est une sous-liste de la suite de S2.

Axiomes_obtenus

```
is_sublist(empty(),empty()) == true()
is_sublist(app(E1,S1),empty()) == false()
is_sublist(app(E1,S1),app(E1,S3)) == is_prefix(S1,S3)
is_sublist(S,app(E2,S3)) == is_sublist(S,S3)

==(is_sublist(empty(),empty()),true())
==(is_sublist(app(E1,S1),empty()),false())
==(is_sublist(app(E1,S1),app(E1,S3)),is_prefix(S1,S3))
==(is_sublist(S,app(E2,S3)),is_sublist(S,S3))

eval(is_sublist(empty(),empty()),Br) <=> Br=true()
eval(is_sublist(app(E1,S1),empty()),Br) <=> Br=false()
eval(is_sublist(app(E1,S1),app(E1,S3)),Br) <=>
                                    eval(is_prefix(S1,S3),Br)
eval(is_sublist(S,app(E2,S3)),Br) <=> eval(is_sublist(S,S3),Br)

eval(is_sublist(S,S2),Br) <= eval(S,empty())  &
                             eval(S2,empty()) & Br=true()
eval(is_sublist(S,S2),Br) <= eval(S,app(E1,S1)) &
                             eval(S2,empty()) & Br=false()
eval(is_sublist(S,S2),Br) <= eval(S,app(E1,S1)) &
            eval(S2,app(E1,S3)) & eval(is_prefix(S1,S3),Br)
eval(is_sublist(S,S2),Br) <= eval(S2,app(E2,S3)) &
                             eval(is_sublist(S,S3),Br)

eval(is_sublist(S,S2),true) :- eval(S,[]), eval(S2,[]).
eval(is_sublist(S,S2),false):- eval(S,[E1|S1]), eval(S2,[]).
eval(is_sublist(S,S2),Br) :- eval(S,[E1,S1]),eval(S2,[E1|S3]),
                             eval(is_prefix(S1,S3),Br).
eval(is_sublist(S,S2),Br) :- eval(S2,[E2|S3]),
                             eval(is_sublist(S,S3),Br)

is_sublist([],[],true).
is_sublist([E1|S1],[],false).
is_sublist([E1,S1],[E1|S3],Br) :- is_prefix(S1,S3,Br).
is_sublist(S,[E2|S3],Br) :- is_sublist(S,S3,Br)
```

fonction is_prefix(S,S2) donne Br.

Avec S,S2 : SEQ,
Br : BOOL.

Cette fonction donne un booléen Br qui est vrai si la liste S est une sous-liste de mêmes éléments, dans le même ordre, de S2, commençant au début de la liste S2.

Paramètre_d'induction S,S2

Relation_bien-fondée

s1‹s2 ssi s1 est un suffixe propre de s2.

Forme_structurelle_du_paramètre_d'induction

S cons1 : empty()
cons2 : app(E1,S1)

S2 cons1 : empty()
cons2 : app(E3,S3)

Construction_des_Fi

- si S est vide, alors S est certainement une sous-séquence préfixe de S2.
- si S n'est pas vide mais que S2 l'est, alors S est une liste plus grande que S2, donc ce n'est pas une sous-séquence préfixe.
- si ni S ni S2 ne sont vides, et que leurs têtes sont les mêmes, il faut vérifier que la suite de S est un préfixe de la suite de S2.
- si ni S ni S2 ne sont vides, et que leurs têtes sont différentes, alors il est certain que S n'est pas une sous-suite préfixe de S2.

Axiomes_obtenus

```
is_prefix(empty(),S2) == true()
is_prefix(S,empty()) == false()
is_prefix(app(E1,S1),app(E1,S3)) == is_prefix(S1,S3)
is_prefix(app(E1,S1),app(E3,S3)) == false()

==(is_prefix(empty(),S2),true())
==(is_prefix(S,empty()),false())
==(is_prefix(app(E1,S1),app(E1,S3)),is_prefix(S1,S3))
==(is_prefix(app(E1,S1),app(E3,S3)),false())
```

```
eval(is_prefix(empty(),S2),Br) <=> Br=true()
eval(is_prefix(S,empty()),Br) <=> Br=false()
eval(is_prefix(app(E1,S1),app(E1,S3)),Br) <=>
                                  eval(is_prefix(S1,S3),Br)
eval(is_prefix(app(E1,S1),app(E3,S3)),Br) <=> Br=false()

eval(is_prefix(S,S2),Br) <= eval(S,empty()) & Br=true()
eval(is_prefix(S,S2),Br) <= eval(S2,empty()) & Br=false()
eval(is_prefix(S,S2),Br) <= eval(S,app(E1,S1)) &
         eval(S2,app(E1,S3)) & eval(is_prefix(S1,S3),Br)
eval(is_prefix(S,S2),Br) <= eval(S,app(E1,S1)) &
                            eval(S2,app(E2,S3)) & Br=false()

eval(is_prefix(S,S2),true) :- eval(S,[]).
eval(is_prefix(S,S2),false) :- eval(S2,[]).
eval(is_prefix(S,S2),Br) :- eval(S,[E1|S1]), eval(S2,[E1|S3]),
                            eval(is_prefix(S1,S3),Br).
eval(is_prefix(S,S2),false) :- eval(S,[E1|S1]),
                               eval(S2,[E2|S3]).

is_prefix([],S2,true).
is_prefix(S,[],false).
is_prefix([E1|S1],[E1|S3],Br) :- is_prefix(S1,S3,Br).
is_prefix([E1|S1],[E2|S3],false).
```

fonction is_ident(S,S2) donne Br.

Avec S,S2 : SEQ,
     Br   : BOOL.

Cette fonction donne un booléen Br qui est vrai si la liste S contient les mêmes éléments, dans le même ordre, que la liste S2.

Paramètre d'induction S,S2

Relation bien-fondée

s1<s2 ssi s1 est un suffixe propre de s2.

Forme structurelle du paramètre d'induction

S	cons1	:	empty()
	cons2	:	app(E1,S1)
S2	cons1	:	empty()
	cons2	:	app(E3,S3)

Construction_des_Fi

- si S et S2 sont vides, c'est qu'ils sont identiques.
- si S et S2 ont la même tête, alors S et S2 sont identiques si leurs suites sont identiques (on refait l'opération de vérification sur les suites de S et S2).
- si S est vide mais que S2 ne l'est pas , alors S est différent de S2. (et inversément).
- si S et S2 ne sont pas vides, et que leurs têtes sont différentes, alors il est certain que S est différent de S2.

Axiomes_obtenus

```
is_ident(empty(),empty()) == true()
is_ident(empty(),app(E1,S1)) == false()
is_ident(app(E1,S1),empty()) == false()
is_ident(app(E1,S1),app(E1,S3)) == is_ident(S1,S3)
is_ident(app(E1,S1),app(E3,S3)) == false()

==(is_ident(empty(),empty(),true())
==(is_ident(empty(),app(E1,S1)),false())
==(is_ident(app(E1,S1),empty()),false())
==(is_ident(app(E1,S1),app(E1,S3)),is_ident(S1,S3)
==(is_ident(app(E1,S1),app(E3,S3)),false())

eval(is_ident(empty(),empty()),Br) <=> Br=true()
eval(is_ident(empty(),app(E1,S1)),Br) <=> Br=false()
eval(is_ident(app(E1,S1),empty()),Br) <=> Br=false()
eval(is_ident(app(E1,S1),app(E1,S3)),Br) <=>
                                  eval(is_ident(S1,S3),Br)
eval(is_ident(app(E1,S1),app(E2,S2)),Br) <=> Br=false()

eval(is_ident(S,S2),Br) <= eval(S,empty()) &
                           eval(S2,empty()) & Br=true()
eval(is_ident(S,S2),Br) <= eval(S,empty()) &
                           eval(S2,app(E1,S1)) & Br=false()
eval(is_ident(S,S2),Br) <= eval(S,app(E1,S1)) &
                           eval(S2,empty()) & Br=false()
eval(is_ident(S,S2),Br) <= eval(S,app(E1,S1)) &
                           eval(S2,app(E2,S2)) &
                           eval(is_ident(S1,S3),Br)
eval(is_ident(S,S2),Br) <= eval(S,app(E1,S1)) &
                           eval(S2,app(E2,S2)) & Br=false()

eval(is_ident(S,S2),true) :- eval(S,[]), eval(S2,[]).
eval(is_ident(S,S2),false) :- eval(S,[]),
                              eval(S2,[E1¦S1]).
eval(is_ident(S,S2),false) :- eval(S,[E1¦S1]),
                              eval(S2,[]).
eval(is_ident(S,S2),Br) :- eval(S,[E1¦S1]),eval(S2,[E2¦S2]),
                           eval(is_ident(S1,S3),Br).
eval(is_ident(S,S2),false) :- eval(S,[E1¦S1]),
                              eval(S2,[E2¦S2]).
```

```
is_ident([],[],true).
is_ident([],[E1|S1],false).
is_ident([E1|S1],[],false).
is_ident([E1|S1],[E2|S2],Br) :- is_ident(S1,S3,Br).
is_ident([E1|S1],[E2|S2],false).
```

Récapitulatif des axiomes ADTs obtenus

```
conc(S1,S2)

     conc(empty(),S2)  == S2
     conc(app(E,S),S2) == app(E,conc(S,S2))

insrt(E,N,S)

     insrt(E,succ(zero()),S) == app(E,S)
     insrt(E,succ(N),app(E2,S2)) == app(E2,insrt(E,N,S2))

del(N,S)

     del(succ(zero()),app(E,S2)) == S2
     del(succ(N),app(E,S2))      == app(E,del(N,S2))

efface(E,S)

     efface(E,empty())    == empty()
     efface(E,app(E,S2))  == S2
     efface(E,app(E2,S2)) == app(E2,efface(E,S2))

eff_all(E,S)

     eff_all(E,empty())    == empty()
     eff_all(E,app(E,S2))  == eff_all(E,S2)
     eff_all(E,app(E2,S2)) == app(E2,eff_all(E,S2))

subseq(N,N2,S)

     subseq(succ(zero()),succ(zero()),app(E,S2)) ==
            app(E,empty())
     subseq(succ(zero()),N2,app(E,S2)) ==
            app(E,subseq(succ(zero()),pred(N2),S2))
     subseq(N,N2,app(E,S2)) == subseq(pred(N),pred(N2),S2)

tail(S)

     tail(app(E,S2)) == S2

reverse(S)

     reverse(empty())   == empty()
     reverse(app(E,S2)) == conc(reverse(S2),app(E,empty))

linear(S)

     linear(empty()) == empty()
     if is_list(E) then
        linear(app(E,S2)) == conc(linear(E),linear(S))
     if not(is_list(E)) then
        linear(app(E,S2)) == conc(app(E,empty),linear(S))
```

```
sort(S,P)

     sort(empty(),P)    == empty()
     sort(app(E,S2),P) == insert(E,sort(S2,P),P)


filter(S,P)

                        filter(empty(),P) == empty()
     if p(E)       then filter(app(E,S2),P) == app(E,filter(S2,P))
     if not(p(E)) then filter(app(E,S2),P) == filter(S2,P)

merge(S,S2,P)

          merge(empty(),S2,P) == S2
          merge(S,empty(),P)  == S
     if p(E)<=p(E2) then
         merge(app(E,S3),app(E2,S4),P) == app(E2,merge(S,S4,P))
     if p(E)>p(E2)  then
         merge(app(E,S3),app(E2,S4),P) == app(E,merge(S3,S2,P))

ith(N,S)

     ith(succ(zero()),app(E,S2)) == E
     ith(N,app(E,S2))             == ith(pred(N),S2)

head(S)

     head(app(E,S2)) == E

last(S)

     last(app(E,empty())) == E
     last(app(E,S2))      == last(S2)

length(S)

     length(empty())   == zero()
     length(app(E,S2)) == succ(length(S2))

is_in(E,S)

     is_in(E,empty())    == false()
     is_in(E,app(E,S2))  == true()
     is_in(E,app(E2,S2)) == is_in(E,S2)

is_empty(S)

     is_empty(empty())    == true()
     is_empty(app(E,S2))  == false()

is_sort(S,P)

     is_sort(empty()) == true()
     is_sort(app(E,empty())) == true()
     if p(E)<=p(E2) then
          is_sort(app(E,app(E2,S2)) == is_sort(app(E2,S2))
     if p(E)>p(E2) then
          is_sort(app(E,app(E2,S2)) == false()
```

```
insert(E,S,P)

     insert(E,empty(),P) == app(E,empty())
     if p(E)<=p(E2) then
         insert(E,app(E2,S2),P) == app(E2,insert(E,S2,P))
     if p(E)>p(E2) then
         insert(E,app(E2,S2),P) == app(E,insert(E2,S2,P))

correct(E,N,S)

     correct(E,zero,S)    ==  ERREUR "position invalide,  hors des
                              limites"
     correct(E,succ(zero()),empty())== ERREUR "position invalide,
                                       hors des limites"
     correct(E,succ(zero()),app(E2,S2)) == app(E,S2)
     correct(E,succ(N1),app(E2,S2)) == app(E2,correct(E,N1,S2))

subst_all(E,E2,S)

     subst_all(E,E2,empty()) == empty()
     subst_all(E,E2,app(E,S2)) == app(E2,subst_all(E,E2,S2))
     subst_all(E,E2,app(E3,S2)) == app(E3,subst_all(E,E2,S2))

subst_one(E,E2,S)

     subst_one(E,E2,empty()) == empty()
     subst_one(E,E2,app(E,S2)) == app(E2,S2)
     subst_one(E,E2,app(E3,S2)) == app(E3,subst_one(E,E2,S2))

subcut(E,E2,S)

     subcut(E,E2,empty()) == empty()
     subcut(E,E2,app(E,S2)) == app(E,prefcut(E2,S2))
     subcut(E,E2,app(E3,S2)) == subcut(E,E2,S2)

prefcut(E,S)

     prefcut(E,empty()) == empty()
     prefcut(E,app(E,S2)) == app(E,empty())
     prefcut(E,app(E2,S2)) == app(E2,prefcut(E,S2))

newseq(E,N)

     newseq(E,zero()) == empty()
     newseq(E,succ(N1)) == app(E,newseq(E,N1))

times(E,S)

     times(E,empty()) == zero()
     times(E,app(E,S2)) == succ(times(E,S2))
     times(E,app(E2,S2)) == times(E,S2)

pos(E,S)

     pos(E,empty()) == zero()
     pos(E,app(E,S2)) == succ(zero())
     pos(E,app(E2,S2)) == succ(pos(E,S2))
```

```
is_at(E,N,S)

     is_at(E,zero,S) == ERREUR "valeur de paramètre fausse"
     is_at(E,succ(zero()),app(E,S2)) == true()
     is_at(E,succ(N1),empty()) == false()
     is_at(E,succ(N1),app(E2,S2)) == is_at(E,N1,S2)

is_sublist(S,S2)

     is_sublist(empty(),empty()) == true()
     is_sublist(app(E1,S1),empty()) == false()
     is_sublist(app(E1,S1),app(E1,S3)) == is_prefix(S1,S3)
     is_sublist(S,app(E2,S3)) == is_sublist(S,S3)

is_prefix(S,S2)

     is_prefix(empty(),S2) == true()
     is_prefix(S,empty()) == false()
     is_prefix(app(E1,S1),app(E1,S3)) == is_prefix(S1,S3)
     is_prefix(app(E1,S1),app(E3,S3)) == false()

is_ident(S,S2)

     is_ident(empty(),empty()) == true()
     is_ident(empty(),app(E1,S1)) == false()
     is_ident(app(E1,S1),empty()) == false()
     is_ident(app(E1,S1),app(E1,S3)) == is_ident(S1,S3)
     is_ident(app(E1,S1),app(E3,S3)) == false()
```

Récapitulatif des procédures PROLOG avec évaluateur

```
conc(S1,S2)

     eval(conc(S,S2),R):- eval(S,[]),eval(S2,R).
     eval(conc(S,S2),[E|I]) :- eval(S,[E|S1]),
                               eval(conc(S1,S2),I).
insrt(E,N,S)

     eval(insrt(E,N,S),[E|S]) :- eval(N,1).
     eval(insrt(E,N,S),[E2|R]) :- eval(S,[E2|S2]),eval(N,N1),
                  N2 is N1-1, eval(insrt(E,N2,S2),R).

del(N,S)

     eval(del(N,S),R)      :- eval(N,1),eval(S,[E|R]),
     eval(del(N,S),[E|R]) :- eval(S,[E|S2]),eval(N,N2),
                             N1 is N2-1, eval(del(N1,S2)),R).
efface(E,S)

     eval(efface(E,S),[]) :- eval(S,[]).
     eval(efface(E,S),R)  :- eval(S,[E|S2]),eval(S2,R).
     eval(efface(E,S),[E2|I]) :- eval(S,[E2|S2]),
                                 eval(efface(E,S2),I).

eff_all(E,S)

     eval(eff_all(E,S),[]) :- eval(S,[]).
     eval(eff_all(E,S),R)  :- eval(S,[E|S2]),
                              eval(eff_all(E,S2),R).
     eval(eff_all(E,S),[E2|I]) :- eval(S,[E2|S2]),
                                  eval(eff_all(E,S2),I).

subseq(N,N2,S)

     eval(subseq(N,N2,S),[E]) :- eval(N,1),eval(N2,1),
                                 eval(S,[E|S2].
     eval(subseq(N,N2,S),[E|I]) :- eval(N,1),eval(S,[E|S2]),
                              eval(subseq(1,pred(N2),S2),I).
     eval(subseq(N,N2,S),R) :- eval(S,[E|S2]),
                       eval(subseq(pred(N),pred(N2),S2),R).

tail(S)

     eval(tail(S),R) :- eval(S,[E|S2]),eval(S2,R).

reverse(S)

     eval(reverse(S),[]) :- eval(S,[]).
     eval(reverse(S),R) :- eval(S,[E|S2]),
                           eval(conc(reverse(S2),[E]),R).
```

```
linear(S)

     eval(linear(S),[]) :- eval(S,[]).
     eval(linear(S),R) :- eval(S,[E|S2]),is_list(E),
                          eval(conc(linear(E),linear(S)),R).
     eval(linear(S),R) :- eval(S,[E|S2]),not(is_list(E)),
                          eval(conc([E],linear(S)),R).

sort(S,P)

     eval(sort(S,P),[]) :- eval(S,[]).
     eval(sort(S,P),R) :- eval(S,[E|S2]),
                          eval(insert(E,sort(S2,P),P),R).

filter(S,P)

     eval(filter(S,P),[])     :- eval(S,[]).
     eval(filter(S,P),[E|I]) :- eval(S,[E|S2]),p(E),
                                eval(filter(S2,P),I).
     eval(filter(S,P),R)      :- eval(S,[E|S2]),not(p(E)),
                                eval(filter(S2,P),R).

merge(S,S2,P)

     eval(merge(S,S2,P),R) :- eval(S,[]),eval(S2,R).
     eval(merge(S,S2,P),R) :- eval(S2,[]),eval(S,R).
     eval(merge(S,S2,P),[E2|I]) :- eval(S,[E|S3]),
                                   eval(S2,[E2|S4]),p(E)<=p(E2),
                                   eval(merge(S,S4,P),I).
     eval(merge(S,S2,P),[E|I])  :- eval(S,[E|S3]),
                                   eval(S2,[E2|S4]),p(E)>p(E2),
                                   eval(merge(S3,S2,P),I).

ith(N,S)

     eval(ith(N,S),R) :- eval(N,1),eval(S,[E|S2]),eval(E,R).
     eval(ith(N,S),R) :- eval(S,[E|S2]),
                         eval(ith(pred(N),S2),R).

head(S)

     eval(head(S),R) :- eval(S,S1),head(S1,R).

last(S)

     eval(last(S),R) :- eval(S,[E]),eval(E,R).
     eval(last(S),R) :- eval(S,[E|S2]),eval(last(S2),R).

length(S)

     eval(length(S),0) :- eval(S,[]).
     eval(length(S),R) :- eval(S,[E|S2]),eval(length(S2),L),
                          R is L+1.
```

```
is_in(E,S)

     eval(is_in(E,S),false) :- eval(S,[]).
     eval(is_in(E,S),true) :- eval(S,[E|S2]).
     eval(is_in(E,S),R) :- eval(S,[E2|S2]),eval(is_in(E,S2),R).

is_empty(S)

     eval(is_empty(S),true) :- eval(S,[]).
     eval(is_empty(S),false) :- eval(S,[E|S2]).

is_sort(S,P)

     eval(is_sort(S,P),true) :- eval(S,[]).
     eval(is_sort(S,P),true) :- eval(S,[E]).
     eval(is_sort(S,P),R) :- eval(S,[E|[E2|S2]]),p(E)<=p(E2),
                             eval(is_sort([E2|S2],P),R).
     eval(is_sort(S,P),false) :- eval(S,[E|[E2|S2]]),p(E)>p(E2).

insert(E,S,P)

     eval(insert(E,S,P),[E]) :- eval(S,[]).
     eval(insert(E,S,P),[E2|I]) :- eval(S,[E2|S2]),p(E)<=p(E2),
                                   eval(insert(E,S2,P),I).
     eval(insert(E,S,P),[E|I]) :- eval(S,[E2|S2]),p(E)>p(E2),
                                  eval(insert(E2,S2,P),I).

correct(E,N,S)

     eval(correct(E,N,S),[E|S2]) :- eval(N,N1), N1 is 1,
                                    eval(S,[E2|S2]).
     eval(correct(E,N,S),[E2|Sr]) :- eval(N,N2), N1 is N2-1,
                                     eval(S,[E2|S2]),
                                     eval(correct(E,N1,S2),Sr).
subst_all(E,E1,S)

     eval(subst_all(E,E2,S),[]) :- eval(S,[]).
     eval(subst_all(E,E2,S),[E2|Sr]) :- eval(S,[E|S2]),
                                        eval(subst_all(E,E2,S2),Sr).
     eval(subst_all(E,E2,S),[E3|Sr]) :- eval(S,[E3|S2]),
                                        eval(subst_all(E,E2,S2),Sr).

subst_one(E,E1,S)

     eval(subst_one(E,E2,S),[]) :- eval(S,[]).
     eval(subst_one(E,E2,S),[E2|S2]) :- eval(S,[E|S2]).
     eval(subst_one(E,E2,S),[E3|Sr]) :- eval(S,[E3|S2]),
                                        eval(subst_one(E,E2,S2),Sr).

subcut(E,E2,S)

     eval(subcut(E,E2,S),[]) :- eval(S,[]).
     eval(subcut(E,E2,S),[E|Sr]) :- eval(S,[E|S2]),
                                    eval(prefcut(E2,S2),Sr).
     eval(subcut(E,E2,S),Sr) :- eval(S,[E3|S2]),
                                eval(subcut(E,E2,S2),Sr).
```

```
prefcut(E,S)

     eval(prefcut(E,S),[]) :- eval(S,[]).
     eval(prefcut(E,S),[E]) :- eval(S,[E|S2]).
     eval(prefcut(E,S),[E2|Sr]) :- eval(S,[E2|S2]),
                                   eval(prefcut(E,S2),Sr).

newseq(E,N)

     eval(newseq(E,N),[]) :- eval(N,0).
     eval(newseq(E,N),[E|Sr]) :- eval(N,N2), N1 is N2-1,
                                 eval(newseq(E,N1),Sr).

times(E,S)

     eval(times(E,S),0) :- eval(S,[]).
     eval(times(E,S),Nr) :- eval(S,[E|S2]),
                            eval(times(E,S2),N1), Nr is N1 + 1.
     eval(times(E,S),Nr) :- eval(S,[E2|S2]),
                            eval(times(E,S2),Nr).

pos(E,S)

     eval(pos(E,S),0) :- eval(S,[]).
     eval(pos(E,S),1) :- eval(S,[E|S2]).
     eval(pos(E,S),Nr) :- eval(S,[E2|S2]),
                          eval(pos(E,S2),N2), Nr is N2+1.

is_at(E,N,S)

     eval(is_at(E,N,S),true) :- eval(N,1), eval(S,[E|S2]).
     eval(is_at(E,N,S),Br) :- eval(N,N2), N1 is N2 -1,
                      eval(S,[E2|S2]), eval(is_at(E,N1,S2),Br).

is_sublist(S,S2)

     eval(is_sublist(S,S2),true) :- eval(S,[]), eval(S2,[]).
     eval(is_sublist(S,S2),false):- eval(S,[E1|S1]), eval(S2,[]).
     eval(is_sublist(S,S2),Br) :- eval(S,[E1,S1]),eval(S2,[E1|S3]),
                                  eval(is_prefix(S1,S3),Br).
     eval(is_sublist(S,S2),Br) :- eval(S2,[E2|S3]),
                                  eval(is_sublist(S,S3),Br)

is_prefix(S,S2)

     eval(is_prefix(S,S2),true) :- eval(S,[]).
     eval(is_prefix(S,S2),false) :- eval(S2,[]).
     eval(is_prefix(S,S2),Br) :- eval(S,[E1|S1]), eval(S2,[E1|S3]),
                                 eval(is_prefix(S1,S3),Br).
     eval(is_prefix(S,S2),false) :- eval(S,[E1|S1]),
                                    eval(S2,[E2|S3]).
```

```
is_ident(S,S2)

     eval(is_ident(S,S2),true) :- eval(S,[]), eval(S2,[]).
     eval(is_ident(S,S2),false) :- eval(S,[]),
                                   eval(S2,[E1|S1]).
     eval(is_ident(S,S2),false) :- eval(S,[E1|S1]),
                                   eval(S2,[]).
     eval(is_ident(S,S2),Br) :- eval(S,[E1|S1]),eval(S2,[E2|S2]),
                                eval(is_ident(S1,S3),Br).
     eval(is_ident(S,S2),false) :- eval(S,[E1|S1]),
                                   eval(S2,[E2|S2]).
```

Récapitulatif des procédures PROLOG sans évaluateur

```
     conc([],S2,S2).
     conc([E|S1],S2,[E|I]) :- conc(S1,S2,I).

     insrt(E,1,S,[E|S]).
     insrt(E,N,[E2|S2],[E2|R]) :- N2 is N-1, insrt(E,N2,S2,R).

     del(1,[_|S2],S2).
     del(N,[E|S2],[E|R]) :- N1=N-1,del(N1,S2,R).

     efface(_,[],[]).
     efface(E,[E|S2],S2).
     efface(E,[E2|S2],[E2|I]) :- efface(E,S2,I).

     eff_all(_,[],[]).
     eff_all(E,[E|S2],R) :- eff_all(E,S2,R).
     eff_all(E,[E2|S2],[E2|I]) :- eff_all(E,S2,I).

     subseq(1,1,[E|_],[E]).
     subseq(1,N2,[E|S2],[E|I]) :- N3 is N2-1, subseq(1,N3,S2,I).
     subseq(N,N2,[_|S2],R) :- N1 is N-1,N3 is N2-1,
                              subseq(N1,N3,S2,R).

     tail([_|S2],S2).

     reverse([],[]).
     reverse([E|S2],R) :- reverse(S2,I),conc(I,[E],R).

     linear([],[]).
     linear([E|S2],R) :- is_list(E),linear(E,E1),
                         linear([E|S2],S1),conc(E1,S1,R).
     linear([E|S2],R) :- not(is_list(E)),linear([E|S2],S1),
                         conc([E],S1,R).

     sort([],_,[]).
     sort([E|S2],P,R) :- sort(S2,P,R1),insert(E,R1,R).

     filter([],_,[]).
     filter([E|S2],P,[E|I]) :- p(E),filter(S2,P,I).
     filter([E|S2],P,R) :- not(p(E)),filter(S2,P,R).
```

```
merge([],S2,_,S2).
merge(S,[],_,S).
merge([E|S],[E2|S4],P,[E2|I]) :- p(E)<=p(E2),
                                 merge([E|S],S4,P,I).
merge([E|S3],[E2|S4],P,[E|I]) :- p(E)>p(E2),
                                 merge(S3,[E2|S4],P,I).

ith(1,[E|_],E).
ith(N,[_|S2],R) :- N2=N-1,ith(N2,S2,R).

head([E|_],E).

last([E],E).
last([_|S2],R) :- last(S2,R).

length([],0).
length([E|S2],R) :- length(S2,R2),R=R2+1.

is_in(_,[],false).
is_in(E,[E|_],true).
is_in(E,[_|S2],R) :- is_in(E,S2,R).

is_empty([],true).
is_empty([_|_],false).

is_sort([],_,true).
is_sort([E],_,true).
is_sort([E|[E2|S2]],P,R) :- p(E)<=p(E2),
                            is_sort([E2|S2],P,R).
is_sort([E|[E2|S2]],P,false) :- p(E)>p(E2).

insert(E,[],_,[E]).
insert(E,[E2|S2],P,[E2|I]) :- p(E)<=p(E2), insert(E,S2,P,I).
insert(E,[E2|S2],P,[E|I])  :- p(E)>p(E2), insert(E2,S2,P,I).

correct(E,1,[_|S2],[E|S2]).
correct(E,N,[E2|S2],[E2|Sr]) :- N1 is N-1, correct(E,N1,S2,Sr).

subst_all(E,E2,[],[]).
subst_all(E,E2,[E|S2],[E2|Sr]) :- subst_all(E,E2,S2,Sr).
subst_all(E,E2,[E3|S2],[E3|Sr]) :- subst_all(E,E2,S2,Sr).

subst_one(E,E2,[],[]).
subst_one(E,E2,[E|S2],[E2|S2]).
subst_one(E,E2,[E3|S2],[E3|Sr]) :- subst_one(E,E2,S2,Sr).

subcut(E,E2,[],[]).
subcut(E,E2,[E|S2],[E|Sr]) :- prefcut(E2,S2,Sr).
subcut(E,E2,[E3|S2],Sr) :- subcut(E,E2,S2,Sr).

prefcut(E,[],[]).
prefcut(E,[E|S2],[E]).
prefcut(E,[E2|S2],[E2|Sr]) :- prefcut(E,S2,Sr).

newseq(E,0,[]).
newseq(E,N,[E|Sr]) :- N1 is N-1, newseq(E,N1,Sr).
```

```
times(E,[],0).
times(E,[E|S2],Nr) :- times(E,S2,N1), Nr is N1 + 1.
times(E,[E2|S2],Nr) :- times(E,S2,Nr).

pos(E,[],0).
pos(E,[E|S2],1).
pos(E,[E2|S2],Nr) :- pos(E,S2,N2), Nr is N2+1.

is_at(E,1,[E|S2],true).
is_at(E,N,[E2|S2],Br) :- N1 is N -1, is_at(E,N1,S2,Br).

is_sublist([],[],true).
is_sublist([E1|S1],[],false).
is_sublist([E1,S1],[E1|S3],Br) :- is_prefix(S1,S3,Br).
is_sublist(S,[E2|S3],Br) :- is_sublist(S,S3,Br)

is_prefix([],S2,true).
is_prefix(S,[],false).
is_prefix([E1|S1],[E1|S3],Br) :- is_prefix(S1,S3,Br).
is_prefix([E1|S1],[E2|S3],false).

is_ident([],[],true).
is_ident([],[E1|S1],false).
is_ident([E1|S1],[],false).
is_ident([E1|S1],[E2|S2],Br) :- is_ident(S1,S3,Br).
is_ident([E1|S1],[E2|S2],false).
```

Procédures PROLOG sans évaluateur, sous forme non relationnelles exprimant les foncteurs retournant une valeur booléenne

```
is_in(E,[E|_]).
is_in(E,[_|S2]) :- is_in(E,S2).

is_empty([],true).

is_sort([],_).
is_sort([E],_).
is_sort([E|[E2|S2]],P) :- p(E)<=p(E2),
                          is_sort([E2|S2],P).

is_at(E,1,[E|S2]).
is_at(E,N,[E2|S2]) :- N1 is N -1, is_at(E,N1,S2).

is_sublist([],[]).
is_sublist([E1,S1],[E1|S3]) :- is_prefix(S1,S3).
is_sublist(S,[E2|S3]) :- is_sublist(S,S3)

is_prefix([],S2).
is_prefix([E1|S1],[E1|S3]) :- is_prefix(S1,S3).

is_ident([],[]).
is_ident([E1|S1],[E2|S2]) :- is_ident(S1,S3).
```

MODULE UNION DISJOINTE

TYPE DE DONNEES : UNION DISJOINTE N° 1

Paramétré par T1, ..., Tn.

Inf : Un objet du type UNION DISJOINTE des n types T1,...,Tn est un objet d'un et d'un seul des types apparaissant dans l'union.

Set : UNION, T1, ..., Tn.

Sigma :

Cons	: inj_i	: Ti	--›	UNION
Modif	: change_i	: Ti, UNION	--›	UNION
Sélec	: value_i	: UNION	--›	Ti
	is_in_i	: Ti,UNION	--›	BOOL
	is_of_i	: UNION	--›	BOOL

Explicatif du rôle des fonctions

inj_i : crée une occurence de l'union de type Ti

change_i : modifie l'occurence de l'union de type T par une occurence de type Ti

value_i : fournit l'occurence de l'union, si cette occurence est de type Ti

is_in_i : Détermine si l'occurence de type Ti est l'occurence de l'union

is_of_i : Détermine si l'occurence de l'union est de type Ti

Remarques :

1.- Chaque foncteur possède dans son nom un identificateur de type qui lui est propre et qui permet de déterminer le type de l'élément qui est contenu dans l'union. Il porte ici l'indice _i, et sera instancié au nom de l'opération portant sur ce type. De ce fait, il n'y a pas un constructeur, mais bien n constructeurs, l'union étant composée des types T1 à Tn.

2.- Afin de tenter la généralisation de ces n opérations en une seule, nous permettrons à l'identificateur de type de devenir partie intégrante des arguments d'une opération. Toutefois, il ne pourra être question de le sélectionner en tant qu'élément d'un type quelconque. - voir seconde version du module UNION -

3.- Puisqu'il y a plusieurs constructeurs de type, et que l'on peut distinguer ces constructeurs en deux catégories, le constructeur du même nom identificateur que l'opération qui l'utilise et ceux qui ont un nom différent, il n'est pas nécessaire de décrire tous les constructeurs de la seconde catégorie, dans la méthode de construction par induction structurelle. Ainsi, on généralisera les constructeurs de nom différent par _j (constructeurs des types (T1 à Tn)\Ti).

Elaboration des spécifications sémantiques

fonction inj_i(T) donne Ur.

Avec : T : Ti,
Ur : UNION.

Cette fonction donne un élément de type UNION qui est de type Ti.

Implémentation :

Prolog est un langage non typé. Cela nous permet de déterminer une union comme étant une séquence de un et un seul élément.

eval(U,inj_i(T)) ›prolog› eval(U,[i,T])

U=inj_i(T) ›prolog› U=[i,T]

fonction change_i(T,U) donne Ur.

Avec : T : Ti,
U,Ur : UNION.

Cette fonction donne un élément de type UNION qui est de type Ti. L'élément de type Tj qui composait l'union précédemment est détruit.

Axiomes_obtenus

```
change_i(T,inj_i(T2)) == inj_i(T)
change_i(T,inj_j(T2)) == inj_i(T)

==(change_i(T,inj_i(T2)),inj_i(T))
==(change_i(T,inj_j(T2)),inj_i(T))

eval(change_i(T,inj_i(T2)),Ur) <=> Ur=inj_i(T)
eval(change_i(T,inj_j(T2)),Ur) <=> Ur=inj_i(T)

eval(change_i(T,U),Ur) <= eval(U,inj_i(T2)) &
                          Ur=inj_i(T)
eval(change_i(T,U),Ur) <= eval(U,inj_j(T2)) &
                          Ur=inj_i(T)
eval(change_i(T,U),[i,T]) :- eval(U,[i,T2]).
eval(change_i(T,U),[i,T]) :- eval(U,[j,T2]).

change_i(T,[i,T2],[i,T]).
change_i(T,[j,T2],[i,T]).

change_i(T,[_],[i,T]).
```

fonction value_i(U) donne T.

Avec : T : Ti,
U : UNION.

Cette fonction donne un élément de type Ti qui est l'élément contenu dans l'union disjointe U.

Axiomes_obtenus

```
value_i(inj_i(T)) == T
value_i(inj_j(T)) == INDEFINI V ERREUR "Types incompatibles"

==(value_i(inj_i(T)),T)

eval(value_i(inj_i(T)),Tr) <=> eval(T,Tr)
```

```
eval(value_i(U),Tr) <= eval(U,inj_i(T)) & eval(T,Tr)

eval(value_i(U),Tr) :- eval(U,[i,T]), eval(T,Tr).

eval(value_i(U),T) :- eval(U,[i,T]).

value_i([i,T],T).
```

fonction is_in_i(T,U) donne Br.

Avec : T : Ti,
U : UNION.
Br : BOOL.

Cette fonction donne un élément de type BOOLEEN qui est vrai si l'élément T est contenu dans l'union U.

Axiomes obtenus

```
is_in_i(T,inj_i(T))  == true()
is_in_i(T,inj_i(T2)) == false()
is_in_i(T,inj_j(T2)) == false()

==(is_in_i(T,inj_i(T)),true())
==(is_in_i(T,inj_i(T2)),false())
==(is_in_i(T,inj_j(T2)),false())

eval(is_in_i(T,inj_i(T)),Br) <=> Br=true()
eval(is_in_i(T,inj_i(T2)),Br) <=> Br=false()
eval(is_in_i(T,inj_j(T2)),Br) <=> Br=false()

eval(is_in_i(T,U),Br) <= eval(U,inj_i(T)) & Br=true()
eval(is_in_i(T,U),Br) <= eval(U,inj_i(T2)) & Br=false()
eval(is_in_i(T,U),Br) <= eval(U,inj_j(T2)) & Br=false()

eval(is_in_i(T,U),true()) :- eval(U,[i,T]).
eval(is_in_i(T,U),false()) :- eval(U,[i,T2]).
eval(is_in_i(T,U),false()) :- eval(U,[j,T2]).

is_in_i(T,[i,T],true).
is_in_i(T,[i,T2],false).
is_in_i(T,[j,T2],false).

is_in_i(T,[i,T]).
```

fonction is_of_i(U) donne Br.

Avec : Br : BOOL,
U : UNION.

Cette fonction donne un élément de type BOOLEEN qui est vrai si l'élément contenu dans l'union U est de type Ti.

Axiomes obtenus

```
is_of_i(inj_i(T2)) == true()
is_of_i(inj_j(T2)) == false()

==(is_of_i(inj_i(T2)),true())
==(is_of_i(inj_j(T2)),false())

eval(is_of_i(inj_i(T2)),Br) <=> Br=true()
eval(is_of_i(inj_j(T2)),Br) <=> Br=false()

eval(is_of_i(U),Br) <= eval(U,inj_i(T2)) & Br=true()
eval(is_of_i(U),Br) <= eval(U,inj_j(T2)) & Br=false()

eval(is_of_i(U),true()) :- eval(U,[i,T2]).
eval(is_of_i(U),false()) :- eval(U,[j,T2]).

is_of_i([i,T2],true).
is_of_i([j,T2],false).
```

MODULE UNION DISJOINTE

TYPE DE DONNEES : UNION DISJOINTE n° 2

Paramétré par T1, ..., Tn.

Inf : Un objet du type UNION DISJOINTE des n types T1,...,Tn est un objet d'un et d'un seul des types apparaissant dans l'union.

Set : UNION, T1, ..., Tn, IDENTIF.

Sigma :

Cons	inj	: IDENTIF, Ti	--› UNION
Modif	change	: IDENTIF, Ti, UNION	--› UNION
Sélec	value	: IDENTIF, UNION	--› Ti
	is_in	: IDENTIF, Ti, UNION	--› BOOL
	is_of	: IDENTIF, UNION	--› BOOL

Explicatif du rôle des fonctions

inj : crée une occurence de l'union de type Ti
change : modifie l'occurence de l'union de type T par une occurence de type Ti
value : fournit l'occurence de l'union, si cette occurence est de type Ti
is_in : Détermine si l'occurence de type Ti est l'occurence de l'union
is_of : Détermine si l'occurence de l'union est de type Ti

Elaboration des spécifications sémantiques

Elaboration des spécifications sémantiques

fonction inj(I,T) donne Ur.

Avec : T : Ti,
I : IDENTIF,
Ur : UNION.

Cette fonction donne un élément de type UNION qui est de type Ti, et qui porte le nom identificateur IDENTIF.

Implémentation :

Prolog est un langage non typé. Cela nous permet de déterminer une union comme étant une séquence de un et un seul élément.

eval(U,inj(I,T)) ›prolog› eval(U,[I,T])

U=inj(I,T) ›prolog› U=[I,T]

fonction change(I,T,U) donne Ur.

Avec : T : Ti,
I : IDENTIF,
U,Ur : UNION.

Cette fonction donne un élément de type UNION qui est de type Ti, et référencé par le nom identificateur IDENTIF. L'élément de type Tj qui composait l'union précédemment est détruit.

Axiomes obtenus

```
change(I,T,inj(I2,T2)) == inj(I,T)

==(change(I,T,inj(I2,T2)),inj(I,T))

eval(change(I,T,inj(I2,T2)),Ur) <=> Ur=inj(I,T)

eval(change(I,T,U),Ur) <= eval(U,inj(I2,T2)) &
                          Ur=inj(I,T)

eval(change(I,T,U),[I,T]) :- eval(U,[I2,T2]).

change(I,T,[I2,T2],[I,T]).

change(I,T,[_],[I,T]).
```

fonction value(I,U) donne T.

Avec : T : Ti,
I : IDENTIF,
U : UNION.

Cette fonction donne un élément de type Ti qui est l'élément contenu dans l'union disjointe U, si le nom identificateur IDENTIF est le même que celui de l'union.

Axiomes_obtenus

```
value(I,inj(I,T)) == T
value(I,inj(I2,T)) == INDEFINI V ERREUR "Types incompatibles"

==(value(I,inj(I,T)),T)

eval(value(I,inj(I,T)),Tr) <=> eval(T,Tr)

eval(value(I,U),Tr) <= eval(U,inj(I,T)) & eval(T,Tr)

eval(value(I,U),Tr) :- eval(U,[I,T]), eval(T,Tr).

eval(value(I,U),T) :- eval(U,[I,T]).

value(I,[I,T],T).
```

fonction is_in(I,T,U) donne Br.

Avec : T : Ti,
I : IDENTIF,
U : UNION.
Br : BOOL.

Cette fonction donne un élément de type BOOLEEN qui est vrai si l'élément T est contenu dans l'union U, et que son nom identificateur de type IDENTIF est le même.

Axiomes_obtenus

```
is_in_i(I,T,inj(I,T))  == true()
is_in_i(I,T,inj(I,T2)) == false()
is_in_i(I,T,inj(I2,T2)) == false()

==(is_in(I,T,inj(I,T)),true())
==(is_in(I,T,inj(I,T2)),false())
==(is_in(I,T,inj(I2,T2)),false())
```

```
eval(is_in(I,T,inj(I,T)),Br) <=> Br=true()
eval(is_in(I,T,inj(I,T2)),Br) <=> Br=false()
eval(is_in(I,T,inj(I2,T2)),Br) <=> Br=false()

eval(is_in(I,T,U),Br) <= eval(U,inj(I,T)) & Br=true()
eval(is_in(I,T,U),Br) <= eval(U,inj(I,T2)) & Br=false()
eval(is_in(I,T,U),Br) <= eval(U,inj(I2,T2)) & Br=false()

eval(is_in(I,T,U),true()) :- eval(U,[I,T]).
eval(is_in(I,T,U),false()) :- eval(U,[I,T2]).
eval(is_in(I,T,U),false()) :- eval(U,[I2,T2]).

is_in(I,T,[I,T],true).
is_in(I,T,[I,T2],false).
is_in(I,T,[I2,T2],false).
```

fonction is_of(I,U) donne Br.

Avec : Br : BOOL,
I : IDENTIF,
U : UNION.

Cette fonction donne un élément de type BOOLEEN qui est vrai si l'élément contenu dans l'union U est de type Ti, c'est-à-dire si le nom idfentificateur IDENTIF est le même que celui de l'union.

Axiomes_obtenus

```
is_of(I,inj(I,T2)) == true()
is_of(I,inj(I2,T2)) == false()

==(is_of(I,inj(I,T2)),true())
==(is_of(I,inj(I2,T2)),false())

eval(is_of(I,inj(I,T2)),Br) <=> Br=true()
eval(is_of(I,inj(I2,T2)),Br) <=> Br=false()

eval(is_of(I,U),Br) <= eval(U,inj(I,T2)) & Br=true()
eval(is_of(I,U),Br) <= eval(U,inj(I2,T2)) & Br=false()

eval(is_of(I,U),true()) :- eval(U,[I,T2]).
eval(is_of(I,U),false()) :- eval(U,[I2,T2]).

is_of(I,[I,T2],true).
is_of(I,[I2,T2],false).
```

Récapitulatif des axiomes ADTs obtenus

```
change(I,T,inj(I2,T2)) == inj(I,T)

value(I,inj(I,T)) == T
value(I,inj(I2,T)) == INDEFINI V ERREUR "Types incompatibles"

is_in_i(I,T,inj(I,T))  == true()
is_in_i(I,T,inj(I,T2)) == false()
is_in_i(I,T,inj(I2,T2)) == false()

is_of(I,inj(I,T2)) == true()
is_of(I,inj(I2,T2)) == false()
```

Récapitulatif des procédures PROLOG avec évaluateur

```
change(I,T,U)

     eval(change(I,T,U),[I,T]) :- eval(U,[I2,T2]).

value(I,U)

     eval(value(I,U),T) :- eval(U,[I,T]).

is_in(I,T,U)

     eval(is_in(I,T,U),true()) :- eval(U,[I,T]).
     eval(is_in(I,T,U),false()) :- eval(U,[I,T2]).
     eval(is_in(I,T,U),false()) :- eval(U,[I2,T2]).

is_of(I,U)

     eval(is_of(I,U),true()) :- eval(U,[I,T2]).
     eval(is_of(I,U),false()) :- eval(U,[I2,T2]).
```

Récapitulatif des procédures PROLOG avec évaluateur

```
change(I,T,[_],[I,T]).

value(I,[I,T],T).

is_in(I,T,[I,T],true).
is_in(I,T,[I,T2],false).
is_in(I,T,[I2,T2],false).

is_of(I,[I,T2],true).
is_of(I,[I2,T2],false).
```

Procédures PROLOG sans évaluateur, sous forme non relationnelles exprimant les foncteurs retournant une valeur booléenne

```
is_in(I,T,[I,T]).

is_of(I,[I,T2]).
```

MODULE PRODUIT CARTESIEN

TYPE DE DONNEES : PC (Produit Cartésien, record, enregistrement)

Paramétré par T1,...,Tn.

Inf : Un objet du type PC de n types est un N-tuple. Chaque composant du N-tuple est caractérisé par un nom de champ et son type correspondant. Les noms de champs doivent être mentionnés explicitement quand plusieurs composants sont de même type.

Set : PC, T1, ..., Tn, BOOL.

Sigma :

Cons : pc_cons: IDENTIF,T1,..., IDENTIF,Tn --› PC

Modif : pc_modify : IDENTIF, Ti, PC --› Ti

Sélec : pc_sel : IDENTIF, PC --› Ti
pc_is_in : IDENTIF, Ti, PC --› BOOL

Remarque :

1.- Un TUPLE est un ensemble d'éléments nommés qui peuvent être de type différent. Il ressemble au Produit Cartésien mathématique, à cela près que ses composants sont référencés par des sélecteurs nommés plutôt que par des indices numériques.

2.- Pour la simplicité des transformations axiomatiques, ainsi que pour la clarté des lignes qui vont suivre, nous avons décidé de réduire à 2 le nombre de types qui composent le produit cartésien. Ce choix n'ajoute en rien à la difficulté, puisqu'il est évident qu'un produit cartésien de plusieurs éléments est un produit cartésien d'un élément et d'un produit cartésien de plusieurs - 1 éléments.

PC[T1,...,Tn] =
PC[T1,PC[T2,...,Tn]] =
PC[T1,PC[T2,PC[T3,...,Tn]]]

Explicatif du rôle des fonctions

pc_cons : Crée une nouvelle occurence d'un produit cartésien de 2 types T1 et T2, référencés par le nom de leur champ.
pc_modify : Modifie la valeur de type Ti d'un champ de nom déterminé d'un produit catésien
pc_sel : Fournit l'élément de type Ti du champ de nom déterminé d'un produit cartésien
pc_is_in : détermine si un élément est bien dans le produit cartésien au champ correspondant à son type

Elaboration des spécifications sémantiques

fonction pc_cons(I1,T1,I2,T2) donne Pr.

Avec : I1,I2 : IDENTIF,
T1 : T1,
T2 : T2,
Pr : PC.

Cette fonction donne un PC qui est constitué de deux champs nommés I1 et I2, et contenant respectivement les éléments T1 de type T1 et T2 de type T2.

Implémentation :

Prolog est un langage non typé. Cela nous permet de déterminer un produit cartésien comm étant une suite de n (dans notre idée de clarté, n = 2) séquences de deux éléments : un nom de champ et la valeur de l'élément du champ.

eval(P,pc_cons(I1,T1,I2,T2)) ›prolog› eval(P,[[I1,T1],[I2,T2]])

P=pc_cons(I1,T1,I2,T2)) ›prolog› P=[[I1,T1],[I2,T2]]

fonction pc_modify(I,T,P) donne Tr.

Avec I : IDENTIF,
T,Tr : Ti (de même type, T1 ou T2),
P : PC[T1,T2].

Cette fonction donne un élément de type PC qui est le produit cartésien P dans lequel on a remplacé l'élément contenu dans le champ référencé par I, par l'élément T.

```
pc_modify(I,T,pc_cons(I,T1,I2,T2)) == pc_cons(I,T,I2,T2)
pc_modify(I,T,pc_cons(I1,T1,I,T2)) == pc_cons(I1,T1,I,T)

==(pc_modify(I,T,pc_cons(I,T1,I2,T2)),pc_cons(I,T,I2,T2))
==(pc_modify(I,T,pc_cons(I1,T1,I,T2)),pc_cons(I1,T1,I,T))

eval(pc_modify(I,T,pc_cons(I,T1,I2,T2)),Pr) <=>
                                    Pr=pc_cons(I,T,I2,T2))
eval(pc_modify(I,T,pc_cons(I1,T1,I,T2)),Pr) <=>
                                    Pr=pc_cons(I1,T1,I,T))

eval(pc_modify(I,T,P),Pr) <= eval(P,pc_cons(I,T1,I2,T2)) &
                             Pr=pc_cons(I,T,I2,T2))
eval(pc_modify(I,T,P),Pr) <= eval(P,pc_cons(I1,T1,I,T2)) &
                             Pr=pc_cons(I1,T1,I,T))


eval(pc_modify(I,T,P),[[I,T],[I2,T2]]) :-
                                  eval(P,[[I,T1],[I2,T2]]).
eval(pc_modify(I,T,P),[[I1,T1],[I,T]]) :-
                                  eval(P,[[I1,T1],[I,T2]]).

pc_modify(I,T,[[I,T1],[I2,T2]],[[I,T],[I2,T2]]).
pc_modify(I,T,[[I1,T1],[I,T2]],[[I1,T1],[I,T]]).
```

fonction pc_sel(I,P) donne Tr.

Avec : I : IDENTIF,
Tr : Ti (type T1 ou T2),
P : PC[T1,T2].

Cette fonction donne un élément de type Ti qui est l'élément contenu dans le champ référencé par I du produit cartésien P.

```
pc_sel(I,pc_cons(I,T1,I2,T2)) == T1
pc_sel(I,pc_concons(I1,T1,I,T2)) == T2
```

```
==(pc_sel(I,pc_cons(I,T1,I2,T2)),T1)
==(pc_sel(I,pc_cons(I1,T1,I,T2)),T2)

eval(pc_sel(I,pc_cons(I,T1,I2,T2)),Tr) <=> eval(T1,Tr)
eval(pc_sel(I,pc_cons(I1,T1,I,T2)),Tr) <=> eval(T2,Tr)

eval(pc_sel(I,P),Tr) <= eval(P,pc_cons(I,T1,I2,T2)) &
                        eval(T1,Tr)
eval(pc_sel(I,P),Tr) <= eval(P,pc_cons(I1,T1,I,T2)) &
                        eval(T2,Tr)

eval(pc_sel(I,P),Tr) :- eval(P,[[I,T1],[I2,T2]]),
                        eval(T1,Tr).
eval(pc_sel(I,P),Tr) :- eval(P,[[I1,T1],[I,T2]]),
                        eval(T2,Tr).

pc_sel(I,[[I,T1],[I2,T2]],T1).
pc_sel(I,[[I1,T1],[I,T2]],T2).
```

fonction pc_is_in(I,T,P) donne Br.

Avec I : IDENTIF,
T : Ti (type T1 ou T2),
P : Pc[T1,T2],
Br : BOOL.

Cette fonction donne un BOOLEEN qui est prend la valeur de vérité Vrai si l'élément T est dans le champ I du produit cartésien P.

```
pc_is_in(I,T,pc_cons(I,T,I2,T2)) == true()
pc_is_in(I,T,pc_cons(I,T1,I2,T2)) == false()
pc_is_in(I,T,pc_cons(I1,T1,I,T)) == true()
pc_is_in(I,T,pc_cons(I1,T1,I,T2)) == false()

==(pc_is_in(I,T,pc_cons(I,T,I2,T2)),true())
==(pc_is_in(I,T,pc_cons(I,T1,I2,T2)),false())
==(pc_is_in(I,T,pc_cons(I1,T1,I,T)),true())
==(pc_is_in(I,T,pc_cons(I1,T1,I,T2)),false())

eval(pc_is_in(I,T,pc_cons(I,T,I2,T2)),Br) <=> Br=true()
eval(pc_is_in(I,T,pc_cons(I,T1,I2,T2)),Br) <=> Br=false()
eval(pc_is_in(I,T,pc_cons(I1,T1,I,T)),Br) <=> Br=true()
eval(pc_is_in(I,T,pc_cons(I1,T1,I,T2)),Br) <=> Br=false()

eval(pc_is_in(I,T,P),Br) <= eval(P,pc_cons(I,T,I2,T2))) &
                            Br=true()
eval(pc_is_in(I,T,P),Br) <= eval(P,pc_cons(I,T1,I2,T2))) &
                            Br=false()
eval(pc_is_in(I,T,P),Br) <= eval(P,pc_cons(I1,T1,I,T))) &
                            Br=true()
eval(pc_is_in(I,T,P),Br) <= eval(P,pc_cons(I1,T1,I,T2))) &
                            Br=false()
```

```
eval(pc_is_in(I,T,P),true) :- eval(P,[[I,T],[I2,T2]]).
eval(pc_is_in(I,T,P),false) :- eval(P,[[I,T1],[I2,T2]]).
eval(pc_is_in(I,T,P),true) :- eval(P,[[I1,T1],[I,T]]).
eval(pc_is_in(I,T,P),false) :- eval(P,[[I1,T1],[I,T2]]).

pc_is_in(I,T,[[I,T],[I2,T2]],true).
pc_is_in(I,T,[[I,T1],[I2,T2]],false).
pc_is_in(I,T,[[I1,T1],[I,T]],true).
pc_is_in(I,T,[[I1,T1],[I,T2]],false).
```

Récapitulatif des axiomes ADTs obtenus

```
pc_modify(I,T,pc_cons(I,T1,I2,T2)) == pc_cons(I,T,I2,T2)
pc_modify(I,T,pc_cons(I1,T1,I,T2)) == pc_cons(I1,T1,I,T)

pc_sel(I,pc_cons(I,T1,I2,T2)) == T1
pc_sel(I,pc_cons(I1,T1,I,T2)) == T2

pc_is_in(I,T,pc_cons(I,T,I2,T2)) == true()
pc_is_in(I,T,pc_cons(I,T1,I2,T2)) == false()
pc_is_in(I,T,pc_cons(I1,T1,I,T)) == true()
pc_is_in(I,T,pc_cons(I1,T1,I,T2)) == false()
```

Récapitulatif des procédures PROLOG avec évaluateur

```
eval(pc_modify(I,T,P),[[I,T],[I2,T2]]) :-
                                  eval(P,[[I,T1],[I2,T2]]).
eval(pc_modify(I,T,P),[[I1,T1],[I,T]]) :-
                                  eval(P,[[I1,T1],[I,T2]]).

eval(pc_sel(I,P),Tr) :- eval(P,[[I,T1],[I2,T2]]),
                        eval(T1,Tr).
eval(pc_sel(I,P),Tr) :- eval(P,[[I1,T1],[I,T2]]),
                        eval(T2,Tr).

eval(pc_is_in(I,T,P),true) :- eval(P,[[I,T],[I2,T2]]).
eval(pc_is_in(I,T,P),false) :- eval(P,[[I,T1],[I2,T2]]).
eval(pc_is_in(I,T,P),true) :- eval(P,[[I1,T1],[I,T]]).
eval(pc_is_in(I,T,P),false) :- eval(P,[[I1,T1],[I,T2]]).
```

Récapitulatif des procédures PROLOG sans évaluateur

```
pc_modify(I,T,[[I,T1],[I2,T2]],[[I,T],[I2,T2]]).
pc_modify(I,T,[[I1,T1],[I,T2]],[[I1,T1],[I,T]]).

pc_sel(I,[[I,T1],[I2,T2]],T1).
pc_sel(I,[[I1,T1],[I,T2]],T2).


pc_is_in(I,T,[[I,T],[I2,T2]],true).
pc_is_in(I,T,[[I,T1],[I2,T2]],false).
pc_is_in(I,T,[[I1,T1],[I,T]],true).
pc_is_in(I,T,[[I1,T1],[I,T2]],false).
```

MODULE PILE

TYPE DE DONNEES : PILE (Stack, tableau variable)

Paramétré par ELEM.

Inf : Un objet du type PILE est une liste d'éléments dont l'ordre est imposé par la loi "Dernier entré, premier sorti" (politique LIFO). Il y a donc héritage des propriétés de la séquence, avec restriction sur l'accès et la modification d'éléments.

Set : STACK, ELEM, BOOL.

Sigma :

Cons	: emptystack	:		--› STACK
	push	:	ELEM, STACK	--› STACK
Modif	: pop	:	STACK	--› STACK
Sélec	: top	:	STACK	--› ELEM

Explicatif du rôle des fonctions

emptystack : Crée une pile ne contenant aucun élément
push : Ajoute un élément au sommet de cette pile

pop : Enlève le dernier élément de la pile

top : Fournit la valeur du dernier élément entré dans la pile

Elaboration des spécifications sémantiques

fonction emptystack() donne Sr.

Avec : Sr : STACK.

Cette fonction donne une pile STACK qui est une suite vide d'éléments.

Implémentation :

eval(S,emptystack()) ›prolog› eval(S,[])

S = emptystack() ›prolog› S = []

Renommage avec réutilisation de la séquence (la pile étant un cas particulier de cette dernière).

emptystack() == empty()

fonction push(E,S) donne Sr.

Avec E : ELEM,
S,Sr : STACK.

Cette fonction donne un élément de type STACK qui est la liste S à laquelle on ajoute au sommet (queue ou début, selon le choix d'impléméentation) l'élément E.

Implémentation :

eval(S,push(E,S1)) ›prolog› eval(S,[E|S1])

S = push(E,S1) ›prolog› S = [E|S1]

Renommage avec réutilisation de la séquence (la pile étant un cas particulier de cette dernière).

push(E,S) == app(E,S)

Donc, les constructeurs sont implémentés de la même manière dans la séquence et dans la pile.

fonction pop(S) donne Sr.

Avec : S,Sr : STACK.

Cette fonction donne une pile Sr qui est la pile S à laquelle on a enlevé l'élément au sommet.

Axiomes obtenus :

pop(emptystack()) == emptystack()
pop(push(E,S)) == S

Renommage avec réutilisation de la séquence (la pile étant un cas particulier de cette dernière).

pop(S) == tail(S)

fonction top(S) donne Er.

Avec : S : STACK,
Er : ELEM.

Cette fonction donne un élément Er qui est l'élément au sommet de la pile S.

Préconditions : la pile S doit avoir au moins un élément
S n'est pas emptystack()

Axiomes obtenus :

top(emptystack()) == INDEFINI ou ERREUR (selon la précondition)
top(push(E,S)) == E

Renommage avec réutilisation de la séquence (la pile étant un cas particulier de cette dernière).

top(S) == head(S)

Récapitulatif des axiomes obtenus

Par_renommage_avec_réutilisation_de_la_séquence .

```
emptystack() == empty()

push(E,S) == app(E,S)

pop(S) == tail(S)

top(S) == head(S)
```

Récapitulatif des procédures PROLOG obtenues, avec évaluateur

Par_renommage_avec_réutilisation_de_la_séquence .

```
eval(emptystack(),Sr) :- eval(empty(),Sr).

eval(push(E,S),Sr) :- eval(app(E,S),Sr).

eval(pop(S),Sr) :- eval(tail(S),Sr).

eval(top(S),Sr) :- eval(head(S),Sr).
```

Récapitulatif des procédures PROLOG obtenues, sans évaluateur

Par_renommage_avec_réutilisation_de_la_séquence .

```
emptystack(Sr) :- empty(Sr).

push(E,S,Sr) :- app(E,S,Sr).

pop(S,Sr) :- tail(S,Sr).

top(S,Sr) :- head(S,Sr).
```

MODULE ENSEMBLE

TYPE DE DONNEES : ENSEMBLE (Set)

Paramétré par ELEM

Inf : Un objet du type ENSEMBLE est une suite d'éléments de même type ELEM, placés dans un ordre quelconque, mais qui ne sont présents qu'en un exemplaire (pas de répétitions de valeur).

Set : SET, ELEM, NAT, BOOL .

Sigma :

Cons	: emptyset	:	--› SET
	appset	: ELEM, SET	--› SET
Modif	: concset	: SET,SET	--› SET
	effset	: ELEM,SET	--› SET
	filterset	: SET,ELEM,PRED	--› SET
	mergeset	: SET,SET,PRED	--› SET
Sélec	: lengthset	: SET	--› NAT
	is_inset	: ELEM,SET	--› BOOL
	is_emptyset	: SET	--› BOOL

Remarque :

1.- Puisque le type ENSEMBLE est défini comme étant une liste sans répétition, celui hérite des opérations du type LISTE, mais avec certaines contraintes :

. il faut vérifier que l'élément à ajouter dans l'ensemble n'y est pas encore (opération is_inset), et

. il faut être certain qu'une modification de valeur d'un élément de l'ensemble n'entraîne une répétition de valeur (il serait simple de dire que, pour modifier la valeur d'un élément, celui-ci doit être enlevé, avant d'ajouter au même endroit la nouvelle valeur).

2.- Certaines opérations sur les listes n'ont aucun intérêt dans le type SET. En effet, faire l'opération reverse qui inverse l'ordre d'une liste est inutile sur un ensemble, l'ordre y étant, par définition, quelconque. Nous donnons, après les explications brèves du rôle des fonctions, la liste des opérations sans intérêt, ou particulières.

3.- Afin de faciliter la manipulation des opérations, nous avons décidé de déterminer un ordre autre que l'ordre d'insertion des éléments dans la liste : l'ordre croissant selon l'opération "plus grand que". Nous changerons ce constructeur de nom (appset devient appsorted).

Explicatif du rôle des fonctions

emptyset : Création d'un ensemble vide.
appset : Ajout d'un élément dans l'ensemble.

concset : Concaténation de deux ensembles.
effset : Enlève l'élément E dans l'ensemble S.
filterset: Filtre un ensemble selon une propriété P commune à tous les éléments.
mergeset : Fusionne deux ensembles selon une propriété commune à tous les éléments.
lengthset: Fournit la longueur d'un ensemble
is_inset : Vérifie l'existence d'un élément dans un ensemble
is_emptyset : Vérifie si l'ensemble est vide

Fonctions de la liste qui n'ont aucun sens pour un ensemble

Les opérations jouant sur la position d'un élément dans une liste n'ont aucun sens pour les ensembles, puisque ceux-ci n'ont pas d'ordre fixé, c'est-à-dire que la position d'un élément n'est pas identificateur de cet élément.

De ce fait, les fonctions suivantes seront manipulées dans les équations des spécifications axiomatiques, mais ne seront pas disponibles aux utilisateurs du module ensemble.

insrt,	del,	subset,	tail,
ith,	head,	last,	eff_all,
reverse,	linear,	sort,	is_sort

```
TYPE DE DONNEES : ENSEMBLE  (Set)                          N° 2

     Modif :  subst_set   :  ELEM,ELEM,SET --› SET

     Sélec :  is_subset   :  SET,SET      --› BOOL
              is_identset:   SET,SET      --› BOOL
```

Explicatif du rôle des fonctions

subst_set : Remplace l'élément de l'ensemble par un autre élément
is_subset : Détermine si un ensemble est un sous-ensemble d'un autre.
is_identset: Détermine si deux ensembles sont identiques

Fonctions de la liste qui n'ont aucun sens pour un ensemble

correct, subst_all, subcut,
prefcut, pos, times,
is_at, is_prefix.

fonction emptyset() donne Sr .

Avec Sr : SET.

Cette fonction donne un ensemble résultant Sr qui est l'ensemble vide.

Postconditions : Sr est l'ensemble vide.

Implémentation :

eval(S,emptyset()) ›prolog› eval(S,[])

S=emptyset() ›prolog› S=[]

Réutilisation du module séquence (dont l'ensemble est un cas particulier)

emptyset() == empty()

fonction appset(E,S) donne Sr.

Avec E : ELEM,
S,Sr : SET.

Cette fonction donne un ensemble résultant Sr qui est l'ensemble S auquel on a ajouté l'élément E.

Réutilisation du module séquence (dont l'ensemble est un cas particulier)

if not(in(E,S)) then appset(E,S) == app(E,S)
if in(E,S) then appset(E,S) == S

Implémentation :

appset(E,emptyset()) == app(E,empty())
appset(E,app(E,S2)) == S2
appset(E,app(E2,S2)) == app(E2,appset(E,S2))

Ici, on ne tient pas compte du choix d'insertion, mais seulement de savoir si un élément est déjà présent ou non dans l'ensemble. Ces axiomes correspondent à la fusion des axiomes déterminant les fonctions app et in de la séquence.

```
appset(E,emptyset()) == app(E,empty())
appset(E,app(E,S2)) == app(E,S2)
appset(E,app(E2,S2)) == app(E2,appset(E,S2))

==(appset(E,emptyset()),app(E,empty()))
==(appset(E,app(E,S2)),app(E,S2))
==(appset(E,app(E2,S2)),app(E2,appset(E,S2)))

eval(appset(E,emptyset()),Sr) <=> Sr=app(E,empty())
eval(appset(E,app(E,S2)),Sr) <=> Sr=app(E,S2)
eval(appset(E,app(E2,S2)),Sr) <=> eval(app(E2,appset(E,S2)),Sr)

eval(appset(E,S),Sr) <= eval(S,emptyset()) & Sr=app(E,empty())
eval(appset(E,S),Sr) <= eval(S,app(E,S2)) & Sr=app(E,S2)
eval(appset(E,S),Sr) <= eval(S,app(E2,S2)) &
                        eval(app(E2,appset(E,S2)),Sr)

eval(appset(E,S),[E]) :- eval(S,[]).
eval(appset(E,S),Sr) :- eval(S,[E|S2]), Sr=[E|S2].
eval(appset(E,S),[E2|Sr]) :- eval(S,[E2|S2]),
                             eval(appset(E,S2),Sr).

appset(E,[],[E]).
appset(E,[E|S2],S2).
appset(E,[E2|S2],[E2|Sr]) :- appset(E,S2,Sr).
```

Comme la contrainte sur l'ensemble n'est valide qu'à l'insertion d'un élément, la lecture du premier élément de l'ensemble est obtenu de la même manière que le premier élément d'une séquence correspondante.

```
eval(S,appset(E,S1))      >prolog>     eval(S,[E|S1])
```

S = appset(E,S1) n'est pas valide puisqu'il existe des axiomes réécrits en fonction du constructeur de la séquence app(E,S).

fonction concset(S1,S2) donne Sr .

Avec S1,S2,Sr : SET,

Cette fonction donne un ensemble résultant Sr qui est la fusion des ensembles S2 à S1.

Axiomes obtenus

```
concset(appset(E,S1),S2) == appset(E,concset(S1,S2))
concset(emptyset(),appset(E,S3))==appset(E,concset(emptyset(),S3))
concset(emptyset(),emptyset()) == emptyset()

==(concset(appset(E,S1),S2),appset(E,concset(S1,S2)))
==(concset(emptyset(),appset(E,S3)),appset(E,concset(emptyset(),S3
==(concset(emptyset(),emptyset()),emptyset())

eval(concset(appset(E,S1),S2),Sr) <=>
                          eval(appset(E,concset(S1,S2)),Sr)
eval(concset(emptyset(),appset(E,S3)),Sr) <=>
                          eval(appset(E,concset(emptyset(),S3)),Sr
eval(concset(emptyset(),emptyset()),Sr) <=> Sr=emptyset()

eval(concset(S,S2),Sr) <= eval(S,appset(E,S1)) &
                          eval(appset(E,concset(S1,S2)),Sr)
eval(concset(S,S2),Sr) <= eval(S,emptyset()) &
                          eval(S2,appset(E,S3)) &
                          eval(appset(E,concset(emptyset(),S3)),S
eval(concset(S,S2),Sr) <= eval(S,emptyset()) &
                          eval(S2,emptyset()) & Sr=emptyset()

eval(concset(S,S2),Sr) :- eval(S,[E|S1]),
                          eval(appset(E,concset(S1,S2)),Sr).
eval(concset(S,S2),Sr) :- eval(S,[]), eval(S2,[E|S3]),
                          eval(appset(E,concset(emptyset(),S3)),S
eval(concset(S,S2),[]) :- eval(S,[]), eval(S2,[]).

concset(S,[E|S1],Sr) :- concset(S1,S2,S4), appset(E,S4,Sr).
concset([],[E|S3],Sr) :- concset([],S3,S4), appset(E,S4,Sr).
concset([],[],[]) :- eval(S,[]), eval(S2,[]).
```

```
fonction effset(E,S) donne Sr .

Avec S,Sr  : SET.
     Er    : ELEM.

Cette fonction donne un ensemble Sr qui est l'ensemble
S auquel on a enlevé l'occurence de l'élément E.
```

axiomes_obtenus

effset(E,S) == efface(E,S)

Fonction de sélection dans un seul ensemble. Il n'y a donc aucune insertion ou modification de valeur. L'ensemble résultant est bien un ensemble et pas une liste générale.

```
fonction filterset(S,P) donne Sr.

Avec P     : PRED,
     S,Sr  : SET.

Cette fonction donne un ensemble Sr qui est le sous-
ensemble des éléments de l'ensemble S, qui satisfont
tous la même propriété P.
```

Axiomes_obtenus

filterset(S,P) == filter(S,P)

Fonction de sélection dans un seul ensemble. Il n'y a donc aucune insertion ou modification de valeur. L'ensemble résultant est bien un ensemble et pas une liste générale.

```
fonction mergeset(S,S2,P) donne Sr.

Avec P          : PRED,
     S,S2,Sr    : SET.

Cette fonction donne un ensemble SR qui résulte de la
fusion des ensembles S et S2 avec un ordre déterminé
par une propriété P.
```

Axiomes_obtenus

```
mergeset(S,S2,P) == concset(filterset(S,P),filterset(S2,P))
```

Cette fonction diffère grandement de l'opération de fusion de la séquence, puisque l'on ne peut considérer un ordre dans les éléments, au niveau conceptuel du moins (on peut le déterminer à l'implémentation pour des raisons d'efficacité).

En fait, on filtre chaque séquence selon la propriété, puis on fait la fusion des sous-ensembles résultant de ces opérations de filtrage.

```
fonction lengthset(S) donne Nr.

Avec Nr     : NAT,
     S      : SET.

Cette fonction donne un entier naturel Nr qui est
le nombre d'éléments de l'ensemble S (ou taille de
l'ensemble).
```

Axiomes_obtenus

```
lengthset(S) == length(S)
```

Fonction de sélection dans un seul ensemble. Il n'y a donc aucune insertion ou modification de valeur. L'ensemble résultant est bien un ensemble et pas une liste générale.

```
fonction is_inset(E,S) donne Br.

Avec Br    : BOOL,
     S     : SET,
     E     : ELEM.

Cette fonction donne une valeur booléenne qui est
vrai si l'élement E appartient à l'ensemble S, faux
sinon.
```

Axiomes_obtenus

```
is_inset(E,S) == is_in(E,S)
```

Fonction de vérification dans un seul ensemble. Il n'y a aucune insertion ou modification de valeur.

```
fonction is_emptyset(S) donne Br.

Avec Br    : BOOL,
     S     : SET,

Cette fonction donne une valeur booléenne qui est
vrai si la liste S est vide, faux sinon
```

Axiomes_obtenus

```
is_emptyset(S) == is_empty(S)
```

Fonction de vérification dans un seul ensemble. Il n'y a aucune insertion ou modification de valeur.

```
fonction subst_set(E,E2,S) donne Sr.

Avec E,E2 : ELEM,
     S,Sr : SET.

Cette fonction donne un ensemble Sr qui est l'ensemble
S dans lequel l'occurence de l'élément E a été changée
par E2.
```

Axiomes_obtenus

```
subst_set(E,E2,emptyset()) == emptyset()
subst_set(E,E2,appset(E,S1)) == appset(E2,S1)
subst_set(E,E2,appset(E3,S1)) == appset(E3,subst_set(E,E2,S1)
```

```
==(subst_set(E,E2,emptyset()),emptyset())
==(subst_set(E,E2,appset(E,S1)),appset(E2,S1))
==(subst_set(E,E2,appset(E3,S1)),appset(E3,subst_set(E,E2,S1))

eval(subst_set(E,E2,emptyset()),Sr) <=> Sr=emptyset()
eval(subst_set(E,E2,appset(E,S1)),Sr) <=>
                          eval(appset(E2,S1),Sr)
eval(subst_set(E,E2,appset(E3,S1)),Sr) <=>
                          eval(appset(E3,subst_set(E,E2,S1),Sr)

eval(subst_set(E,E2,S),Sr) <= eval(S,emptyset()) &
                              Sr=emptyset()
eval(subst_set(E,E2,S),Sr) <= eval(S,appset(E,S1)) &
                              eval(appset(E2,S1),Sr)
eval(subst_set(E,E2,S),Sr) <= eval(S,appset(E3,S1)) &
                           eval(appset(E3,subst_set(E,E2,S1),Sr)

eval(subst_set(E,E2,S),[]) :- eval(S,[]).
eval(subst_set(E,E2,S),Sr) :- eval(S,[E|S1]),
                              eval(appset(E2,S1),Sr).
eval(subst_set(E,E2,S),Sr) :- eval(S,[E3|S1]),
                          eval(appset(E3,subst_set(E,E2,S1),Sr).

subst_set(E,E2,[],[]).
subst_set(E,E2,,[E|S1],Sr) :- appset(E2,S1,Sr).
subst_set(E,E2,S,Sr) :- eval(S,[E3|S1]),
                          eval(appset(E3,subst_set(E,E2,S1),Sr).
```

```
fonction is_subset(S,S2) donne Br.

Avec S,S2 : SET,
     Br   : BOOL.

Cette fonction donne un booléen Br qui est vrai si
l'ensemble S est un sous-ensemble de l'ensemble S2.
```

Le problème de cette opération est qu'il est nécessaire de vérifier que chaque élément du premier ensemble est dans l'ensemble S2. Comme il n'y a pas d'ordre dans l'ensemble, on pourrait trier les deux ensembles au préalable, afin de simplifier les axiomes.

Une autre solution, plus complexe, est de vérifier qu'un élément de S est dans S2, puis d'éliminer cet élément dans les deux ensembles, et de recommencer jusqu'à épuisement d'un des deux. Au cas où un des deux ensembles aurait encore au mojns un élément, il est certain que l'un n'est pas sous-ensemble de l'autre.

Axiomes_obtenus

```
is_subset(emptyset(),emptyset()) == true()
is_subset(emptyset(),appset(E,S1)) == true()
is_subset(appset(E,S1),emptyset()) == false()
is_subset(appset(E,S1),appset(E,S2)) == is_subset(S1,S2)
if is_inset(E,appset(E2,S2)) then
     is_subset(appset(E,S1),appset(E2,S2)) ==
                    is_subset(S1,appset(E2,effset(E,S2)))
```

Ce dernier axiome peut certainement être optimalisé par une réécriture et une utilisation de nouveaux axiomes, plutôt que par l'utilisation de fonctions qui ne retiennent pas certains états que l'on retrouve ultérieurement (telles les opérations is_inset, qui détermine si l'élément E est dans S2, et effset, qui enlève l'élément E de l'ensemble S2).

Toutefois, il ne faut pas perdre de vue que la spécification en types abstraits de données ne décrit que des fonctions unaires. Par là, il est difficile de fusionner deux fonctions aux résultats différents.

```
eval(is_subset(S,S2),true) :- eval(S,[]), eval(S2,[]).
eval(is_subset(S,S2),true) :- eval(S,[]), eval(S2,[E|S1]).
eval(is_subset(S,S2),false) :- eval(S,[E|S1]), eval(S2,[]).
eval(is_subset(S,S2),Br) :- eval(S,[E|S1]),eval(S2,[E|S3]),
                            eval(is_subset(S1,S3),Br).
eval(is_subset(S,S2),Br) :- eval(S2,[E2|S3]),
                            eval(is_inset(E,S3),true),
                            eval(S,[E|S1]),
          eval(is_subset(S1,appset(E2,effset(E,[E2|S3]))),Br).
```

```
is_subset([],[],true).
is_subset([],[_|_],true).
is_subset([_|_],[],false).
is_subset([E|S1],[E|S3],Br) :- is_subset(S1,S3,Br).
is_subset([E|S1],[E2|S3],Br) :- is_inset(E,[E2|S3],true),
                 effset(E,S3,S4), is_subset(S1,[E2|S4],Br).
```

fonction is_identset(S,S2) donne Br.

Avec S,S2 : SET,
     Br   : BOOL.

Cette fonction donne un booléen Br qui est vrai si l'ensemble S contient les mêmes éléments, en même nombre que l'ensemble S2.

Le problème de cette opération est qu'il est nécessaire de vérifier que chaque élément du premier ensemble est dans l'ensemble S2 (même remarque que pour is_subset) et qu'il n'y a que ceux-là.

Un ensemble est identique à un autre si il est un sous-ensemble de cet autre et qu'il comporte le même nombre d'éléments (même taille). Cette spécification est CORRECTE, mais elle est probablement inefficace (optimalisation possible).

Axiomes_obtenus

```
is_identset(S,S2) ==
   and(is_subset(S,S2),is_equal(lengthset(S),lengthset(S2)))
```

Récapitulatif des axiomes obtenus

```
appset(E,emptyset()) == app(E,empty())
appset(E,app(E,S2)) == app(E,S2)
appset(E,app(E2,S2)) == app(E2,appset(E,S2))

concset(appset(E,S1),S2) == appset(E,concset(S1,S2))
concset(emptyset(),appset(E,S3))==appset(E,concset(emptyset(),S3))
concset(emptyset(),emptyset()) == emptyset()

effset(E,S) == efface(E,S)

filterset(S,P) == filter(S,P)

mergeset(S,S2,P) == concset(filterset(S,P),filterset(S2,P))

lengthset(S) == length(S)

is_inset(E,S) == is_in(E,S)

is_emptyset(S) == is_empty(S)

subst_set(E,E2,emptyset()) == emptyset()
subst_set(E,E2,appset(E,S1)) == appset(E2,S1)
subst_set(E,E2,appset(E3,S1)) == appset(E3,subst_set(E,E2,S1)

is_subset(emptyset(),emptyset()) == true()
is_subset(emptyset(),appset(E,S1)) == true()
is_subset(appset(E,S1),emptyset()) == false()
is_subset(appset(E,S1),appset(E,S2)) == is_subset(S1,S2)
if is_inset(E,appset(E2,S2)) then
     is_subset(appset(E,S1),appset(E2,S2)) ==
                    is_subset(S1,appset(E2,effset(E,S2)))

is_identset(S,S2) ==
   and(is_subset(S,S2),is_equal(lengthset(S),lengthset(S2)))
```

Récapitulatif des procédures PROLOG avec évaluateur

```
eval(appset(E,S),[E]) :- eval(S,[]).
eval(appset(E,S),Sr) :- eval(S,[E|S2]), Sr=[E|S2].
eval(appset(E,S),[E2|Sr]) :- eval(S,[E2|S2]),
                             eval(appset(E,S2),Sr).

eval(concset(S,S2),Sr) :- eval(S,[E|S1]),
                          eval(appset(E,concset(S1,S2)),Sr).
eval(concset(S,S2),Sr) :- eval(S,[]), eval(S2,[E|S3]),
                          eval(appset(E,concset(emptyset(),S3)),Sr
eval(concset(S,S2),[]) :- eval(S,[]), eval(S2,[]).

eval(effset(E,S),Sr) :- eval(efface(E,S),Sr).

eval(filterset(S,P),Sr) :- eval(filter(S,P),Sr).

eval(mergeset(S,S2,P),Sr) :-
            eval(concset(filterset(S,P),filterset(S2,P)),Sr)

eval(lengthset(S),Nr) :- eval(length(S),Nr).

eval(is_inset(E,S),Br) :- eval(is_in(E,S),Br).

eval(is_emptyset(S),Br) :- eval(is_empty(S),Br).

eval(subst_set(E,E2,S),[]) :- eval(S,[]).
eval(subst_set(E,E2,S),Sr) :- eval(S,[E|S1]),
                              eval(appset(E2,S1),Sr).
eval(subst_set(E,E2,S),Sr) :- eval(S,[E3|S1]),
                        eval(appset(E3,subst_set(E,E2,S1),Sr).

eval(is_subset(S,S2),true) :- eval(S,[]), eval(S2,[]).
eval(is_subset(S,S2),true) :- eval(S,[]), eval(S2,[E|S1]).
eval(is_subset(S,S2),false) :- eval(S,[E|S1]), eval(S2,[]).
eval(is_subset(S,S2),Br) :- eval(S,[E|S1]),eval(S2,[E|S3]),
                            eval(is_subset(S1,S3),Br).
eval(is_subset(S,S2),Br) :- eval(S2,[E2|S3]),
                            eval(is_inset(E,S3),true),
                            eval(S,[E|S1]),
          eval(is_subset(S1,appset(E2,effset(E,[E2|S3]))),Br).

eval(is_identset(S,S2),Br) :-
       eval(and(is_subset(S,S2),
                is_equal(lengthset(S),lengthset(S2))),Br).
```

Récapitulatif des procédures PROLOG sans évaluateur

```
appset(E,[],[E]).
appset(E,[E|S2],S2).
appset(E,[E2|S2],[E2|Sr]) :- appset(E,S2,Sr).

concset(S,[E|S1],Sr) :- concset(S1,S2,S4), appset(E,S4,Sr).
concset([],[E|S3],Sr) :- concset([],S3,S4), appset(E,S4,Sr).
concset([],[],[]) :- eval(S,[]), eval(S2,[]).

effset(E,S,Sr) :- efface(E,S,Sr).

filterset(S,P,Sr) :- filter(S,P,Sr).

mergeset(S,S2,P),Sr) :- filterset(S,P,S3), filterset(S2,P,S4),
                        concset(S3,S4,Sr).

lengthset(S,Nr) :- length(S,Nr).

is_inset(E,S,Br) :- is_in(E,S,Br).

is_emptyset(S,Br) :- is_empty(S,Br).

subst_set(E,E2,[],[]).
subst_set(E,E2,,[E|S1],Sr) :- appset(E2,S1,Sr).
subst_set(E,E2,S,Sr) :- eval(S,[E3|S1]),
                         eval(appset(E3,subst_set(E,E2,S1),Sr).

is_subset([],[],true).
is_subset([],[_|_],true).
is_subset([_|_],[],false).
is_subset([E|S1],[E|S3],Br) :- is_subset(S1,S3,Br).
is_subset([E|S1],[E2|S3],Br) :- is_inset(E,[E2|S3],true),
                 effset(E,S3,S4), is_subset(S1,[E2|S4],Br).

is_identset(S,S2,Br) :- lengthset(S,N1), lengthset(S2,N2),
                        is_equal(N1,N2,B1), is_subset(S,S2,B2),
                        and(B1,B2,Br).
```

MODULE TABLE

TYPE DE DONNEES : TABLE (Tableau, Vecteur)

Paramétré par ELEM, NAT, NAT.

Inf : Un objet du type TABLE est une séquence d'éléments de type ELEM numérotée de NAT à NAT.

Set : TABLE, ELEM, NAT, BOOL.

Sigma :

Cons	:	alloc	: ELEM, NAT, NAT	--› TABLE
		store	: ELEM, NAT, TABLE	--› TABLE
Sélec	:	fetch	: NAT, TABLE	--› ELEM
		bottom	: TABLE	--› NAT
		top	: TABLE	--› NAT

Explicatif du rôle des fonctions

alloc : Crée un tableau à une dimension dont tous les éléments sont initialisés à E, et numérotés de N1 à N2
store : Modifie la valeur de l'élément à la position N du tableau à une dimension T

fetch : Fournit l'élément à la position N du tableau T
bottom : Fournit la position minimum du tableau T
top : Fournit la position maximum du tableau T

Réutilisation des connaissances

Une table est une séquence d'éléments. On peut réutiliser les axiomes propres à la séquence, avec une contrainte supplémentaire sur la position des éléments dans la séquence, et sur la longueur de cette séquence. Ainsi, la table est une suite de trois éléments de type différent (ou produit cartésien) : la borne inférieure, la borne supérieure et la suite des éléments de cette table.

Spécification par abstraction :

TABLE[ELEM,NAT,NAT] ⟼ PC[Low:NAT, High:NAT, SEQ[ELEM]]

Dans cette optique, les opérations portant sur une table seront des fonctions sur un PC puis sur une séquence.

Elaboration des spécifications sémantiques

fonction alloc(E,N,N2) donne Tr.

Avec : N,N2 : NAT,
E : ELEM,
Tr : TABLE.

Cette fonction donne un tableau à une dimension Tr qui est une suite d'éléments E numérotés de N à N2.

Réutilisation_des_connaissances

```
alloc(E,N,N2) ==
  pc_cons(low,N,high,N2,element,newseq(E,succ(moins(N2,N))))

eval(alloc(E,N,N2),Tr) :- N1 is N2-N+1,
      eval(pc_cons(low,N,high,N2,element,newseq(E,N1)),Tr).

alloc(E,N,N2,Tr) :- N1 is N2-N+1, newseq(E,N1,S),
                    pc_cons(low,N,high,N2,element,S,Tr).
```

Implémentation :

```
eval(T,alloc(E,N,N2))    ›prolog›   eval(T,[N,N2,[E|S]])

T = alloc(E,N,N2)        ›prolog›   T = [N,N2,[E|S]]
```

fonction store(E,N,T) donne Tr.

Avec E : ELEM,
N : NAT,
T,Tr : TABLE.

Cette fonction donne une table Tr qui est identique à la table T, excepté pour l'élément à la position N qui a été remplacé par E.

Réutilisation_des_connaissances :

Changer un élément du tableau revient à modifier l'élément elem (qui est une séquence). Il faut donc sélectionner cette séquence, et changer la valeur de l'élément à la position N-low+1.

```
store(E,N,T) ==
   pc_modify(elem,
             correct(E,
                     succ(moins(N,pc_sel(low,T))),
                     pc_sel(elem,T)),
             T)

eval(store(E,N,T),Tr) :-
   eval(pc_modify(elem,
                  correct(E,
                          succ(moins(N,pc_sel(low,T))),
                          pc_sel(elem,T)),
                  T),
        Tr)

store(E,N,T,Tr) :- pc_sel(elem,T,S), pc_sel(low,T,N1),
                   N2 is N-N1+1, correct(E,N2,S,S2)
                   pc_modify(elem,S2,T,Tr).
```

fonction fetch(N,T) donne Er.

Avec N : NAT,
T : TABLE,
Er : ELEM.

Cette fonction donne un élément Er qui est l'élément localisé à la position N de la table T à une dimension.

Préconditions : N >= bottom(T), N <= top(T)

Axiomes_obtenus :

```
fetch(N,T) ==
          ith(succ(moins(N,pc_sel(low,T))),pc_sel(elem,T))

eval(fetch(N,T),Er) :-
  eval(ith(succ(moins(N,pc_sel(low,T))),pc_sel(elem,T)),Er).

fetch(N,T,Er) :- pc_sel(elem,T,S), pc_sel(low,T,N1),
                 N2 is N-N1+1, ith(N2,S,Er).
```

fonction bottom(T) donne Nr.

Avec Nr : NAT,
T : TABLE.

Cette fonction donne un élément Nr de type naturel qui la valeur de l'indice minimum de la table à une dimension T.

Réutilisation_des_connaissances :

```
bottom(T) == pc_sel(low,T)

eval(bottom(T),Nr) :- eval(pc_sel(low,T),Tr).

bottom(T,Nr) :- pc_sel(low,T,Tr).
```

fonction top(T) donne Nr.

Avec Nr : NAT,
T : TABLE.

Cette fonction donne un élément Nr de type naturel qui la valeur de l'indice maximum de la table à une dimension T.

Réutilisation_des_connaissances :

```
top(T) == pc_sel(high,T)

eval(top(T),Nr) :- eval(pc_sel(high,T),Tr).

top(T,Nr) :- pc_sel(high,T,Tr).
```

Récapitulatif des axiomes obtenus

```
alloc(E,N,N2) ==
  pc_cons(low,N,high,N2,element,newseq(E,succ(moins(N2,N))))

store(E,N,T) ==
   pc_modify(elem,
             correct(E,
                     succ(moins(N,pc_sel(low,T))),
                     pc_sel(elem,T)),
             T)

fetch(N,T) ==
          ith(succ(moins(N,pc_sel(low,T))),pc_sel(elem,T))

bottom(T) == pc_sel(low,T)

top(T) == pc_sel(high,T)
```

Récapitulatif des procédures PROLOG avec évaluateur

```
eval(alloc(E,N,N2),Tr) :- N1 is N2-N+1,
       eval(pc_cons(low,N,high,N2,element,newseq(E,N1)),Tr).

eval(store(E,N,T),Tr) :-
   eval(pc_modify(elem,
                  correct(E,
                          succ(moins(N,pc_sel(low,T))),
                          pc_sel(elem,T)),
                  T),
        Tr)

eval(fetch(N,T),Er) :-
  eval(ith(succ(moins(N,pc_sel(low,T))),pc_sel(elem,T)),Er).

eval(bottom(T),Nr) :- eval(pc_sel(low,T),Tr).

eval(top(T),Nr) :- eval(pc_sel(high,T),Tr).
```

Récapitulatif des procédures PROLOG sans évaluateur

```
alloc(E,N,N2,Tr) :- N1 is N2-N+1, newseq(E,N1,S),
                    pc_cons(low,N,high,N2,element,S,Tr).

store(E,N,T,Tr) :- pc_sel(elem,T,S), pc_sel(low,T,N1),
                   N2 is N-N1+1, correct(E,N2,S,S2)
                   pc_modify(elem,S2,T,Tr).

fetch(N,T,Er) :- pc_sel(elem,T,S), pc_sel(low,T,N1),
                 N2 is N-N1+1, ith(N2,S,Er).

bottom(T,Nr) :- pc_sel(low,T,Tr).

top(T,Nr) :- pc_sel(high,T,Tr).
```